DOCUMENT RESUME

ED 371 739	IR 054 995
AUTHOR	Russell, Dale W.
TITLE	Language Acquisition in a Unification-Based Grammar Processing System Using a Real-World Knowledge Base.
INSTITUTION	Illinois Univ., Urbana. Dept. of Computer Science.
REPORT NO	UILU-ENG-93-1737; UIUCDCS-R-93-1822
PUB DATE	Jul 93
NOTE	392p.; Ph.D Dissertation, University of Illinois at Urbana-Champaign.
PUB TYPE	Dissertations/Theses - Doctoral Dissertations (041) -- Reports - Research/Technical (143)
EDRS PRICE	MF01/PC16 Plus Postage.
DESCRIPTORS	*Computational Linguistics; Computer Oriented Programs; Computer Software Development; Context Effect; Databases; *Definitions; Grammar; Knowledge Level; Language Acquisition; Language Research; *Lexicology; Semantics; *Syntax
IDENTIFIERS	*Natural Language

ABSTRACT

An obstacle in Natural Language understanding is the existence of lexical gaps, i.e. words or word senses that are not in the lexicon of the system. This thesis describes the implementation of MURRAY, a learning mechanism which infers the properties of a new lexical item from its syntactical environment and infers its meaning based on context and a domain-specific database of real-world knowledge. It combines these syntactic and semantic properties to result in a version space of possible lexical entries for the unknown. MURRAY is an extension to an existing unification-based grammar processing system, UNICORN. It is compatible with grammars written in the style of head-driven phrase structure grammar. For each word that does not exist in the lexicon, MURRAY constructs a lexical version hyperspace, and then from multiple inputs the system converges on the target definition of the new world. Included in the five appendixes are sample grammars and entries, learning procedure code, and some sample learning episodes. (Contains 64 references.) (Author/SLD)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED 371 739

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

DCS
THE NEW ADDITION

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)
☐ This document has been reproduced as received from the person or organization originating it
☐ Minor changes have been made to improve reproduction quality
• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy

PORT NO. UIUCDCS-R-93-1822

UILU-ENG-93-1737

Language Acquisition in a Unification-Based Grammar Processing System Using a Real-World Knowledge Base

by

Dale W. Russell

July 1993

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY
J. L. Pence
TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

IR054995

BEST COPY AVAILABLE

LANGUAGE ACQUISITION IN A UNIFICATION-BASED GRAMMAR PROCESSING SYSTEM USING A REAL-WORLD KNOWLEDGE BASE

BY

DALE W. RUSSELL

B.A., Geneva College, 1981
A.M., University of Illinois, 1983
Ph.D., University of Illinois, 1987

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1993

Urbana, Illinois

LANGUAGE ACQUISITION IN A UNIFICATION-BASED GRAMMAR PROCESSING SYSTEM USING A REAL-WORLD KNOWLEDGE BASE

Dale W. Russell, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1993
David C. Wilkins, Advisor

One of the obstacles to be overcome in Natural Language Understanding is the existence of lexical gaps; that is, words or word senses which are not in the lexicon of the system. No lexicon, whether hand-coded or derived from an on-line dictionary, can ever be complete, in the sense of having entries for every word encountered in every syntactic category and with every semantic sense with which it may be used.

In order to address this issue, this thesis describes the implementation of MURRAY, a learning mechanism which is able to (i) infer the syntactic properties of a new lexical item from its syntactic environment; (ii) infer the meaning of a novel lexical item based on context and a domain-specific database of real-world knowledge; and (iii) combine those syntactic and semantic properties of a given unknown word inferred from multiple pieces of input, resulting in a version space of possible lexical entries for the unknown, each consistent with all environments in which it has been encountered.

MURRAY is an extension to an existing unification-based grammar processing system, UNICORN. It has been implemented to operate with grammars written in the style of Head-Driven Phrase Structure Grammar, though it is compatible with any unification-based grammar formalism. On each encounter with a word which does not exist in the lexicon of the system, MURRAY constructs a **lexical version hyperspace**, a disjunction of version spaces, one for each possible syntactic category that the unknown could be in the given linguistic context. On each encounter with the unknown, information from the version spaces thus constructed is combined, so that from multiple inputs, the system converges on the target definition of the new word.

ACKNOWLEDGEMENTS

I would first like to thank my advisor, David C. Wilkins, for initially encouraging me to pursue doctoral work in computer science, and for help and support through the years that followed. I would also like to thank the other members of my thesis committee: Jerry Morgan, Georgia Green, Mehdi Harandi and Simon Kaplan. Special thanks to Jerry and Georgia, for having done double duty, serving on my committees in both computer science and linguistics.

I am especially grateful to Georgia for the prompt and insightful comments she gave on everything that I asked her to read, and for allowing me to participate in her thesis discussion group. The other members of this group provided much helpful discussion and interaction, as well as the welcome opportunity to hear about their fascinating work in linguistics. They are Wen-Ying Lu, Lynne Murphy, Nalini Rau, Sonoko Sakakibara and Timothy Williams. I also benefited greatly from interaction with the members of the Knowledge-Based Systems Research Group: Steve Donoho, Tom Ioerger, Lucja Iwanska, Carl Kadie, Vance Morrison, Ziad Najem, Sheldon Nicholl, Young-Tack Park, and Yong Ma.

Much of what I have learned about working with unification-based grammar processing systems, as well as the motivation behind this thesis work, came from working with the Computational Linguistics group at the Beckman Institute, on a project funded by Motorola, Inc. I greatly appreciate the support that this group has received from the Digital Technology Lab at Motorola, and the helpful interaction we have had with Larry Puhl, Harry Bliss, Tod Farrell and many others. I very much enjoyed working with the past and present members of the Computational Linguistics group: Professors Jerry Morgan, Georgia Green, Erhard Hinrichs and Jennifer Cole, and student members Dale Gerdemann, John Griffith, Molly Homer, Feng-Lan Kuo, Mike Lake, Kang-Hyuk Lee, Rich Scherl and Herry Sutanto. The work I have done here has as its starting point the UNICORN system, created by Dale Gerdemann and Erhard Hinrichs. I would like to thank them both for their help in introducing me to the system, and for allowing me to use it in my thesis work.

Like everyone else in the Cognitive Science Research Group at Beckman, I could not have survived without the invaluable help of Linda May, who supplied everything from cookies to expertise in LaTeX.

Finally, I would like to thank my family, who demonstrated remarkable patience and support throughout my graduate career, and my friends, who have blessed me more than they can ever know.

TABLE OF CONTENTS

Chapter

1 Introduction 1
  1.1 Focus of the Research 1
    1.1.1 The Goal: Understanding All Lexical Items Encountered 2
    1.1.2 Lexical Gaps in Natural Language Text 3
    1.1.3 The Material to be Learned 5
  1.2 Outline of the Thesis 6

2 Relation to Existing Work 8
  2.1 Initial State of the Lexicon 8
    2.1.1 Building the Lexicon from Scratch 9
    2.1.2 Extending a Core Lexicon 13
    2.1.3 Extending a 'Complete' Lexicon 15
  2.2 Method of Learning 19
    2.2.1 Learning by Being Told 20
    2.2.2 Learning from Statistics 24
    2.2.3 Applying General Models of Learning 25
    2.2.4 Learning from Context and Real-World Knowledge 27

3 Theoretical Background 30
  3.1 The Grammatical Theory Underlying HPSG 30
    3.1.1 Context-Free PSGs 31
    3.1.2 Gazdar's GPSG 34
    3.1.3 Pollard and Sag's HPSG 36
  3.2 The UNICORN Parsing System 39
    3.2.1 Earley's Algorithm 39
    3.2.2 Shieber's Extension to the Earley Algorithm 41
    3.2.3 Optimizations in UNICORN 42
  3.3 The Advantages of Unification 42
  3.4 Writing Grammars for UNICORN 44
    3.4.1 Path Equation Notation 44
    3.4.2 Definitions 46

4 Initial Knowledge Used as the Basis for Learning 53
  4.1 The Grammar Used in Learning 54
    4.1.1 The Implementation of HPSG2 in UNICORN 54
    4.1.2 Differences from HPSG2 60

4.1.3 Additions to HPSG2 68
4.2 The Knowledge Base 72
4.2.1 Real-World Knowledge of a Limited Domain 72
4.2.2 Non-Literal Meaning 76
4.2.3 Semantic Checking 77
4.2.4 An Issue of Semantic Representation 78

5 The Semantics of Adverbs and Verbs in HPSG 81
5.1 The Initial Hypothesis: Adverbs vs. Adjectives 82
5.2 Assigning Sorts to Adverbial Modifiers 87
5.3 Attaching Restrictions to Verb Phrase Semantics 90
5.3.1 The Difference List Approach 90
5.3.2 A Modification to the Semantic CONTENT of Verbs 96
5.4 Conclusion 97

6 Hypothesis Construction 99
6.1 Overview 99
6.2 Detecting an Opportunity for Learning 100
6.3 Inferring Syntactic Category 102
6.3.1 Establishing Upper Bounds 102
6.3.2 Establishing Lower Bounds 105
6.4 Inferring Semantics 113
6.4.1 Description of Semantic Knowledge 113
6.4.2 The Method of Acquiring Semantic Knowledge 116
6.4.3 Extensionality vs. Intensionality 120

7 Hypothesis Combination 123
7.1 Combining Upper Bounds 124
7.1.1 Cross-Unification of Upper Bounds 124
7.1.2 Discerning Distinct Lexical Items 127
7.2 Combining Lower Bounds 128
7.3 An Example Learning Episode 130
7.4 Convergence on a Lexical Definition 133
7.5 Computational Complexity 135
7.5.1 Time 135
7.5.2 Space 138

8 Related Issues 140
8.1 Lexical Rules and Their Applications 140
8.1.1 Relating Morphological Forms of a Word 141
8.1.2 Metonymy 144
8.1.3 Metaphor 146
8.1.4 Learning New Syntactic Categories for Previously Known Words 147
8.2 Idioms 156
8.3 Storing the New Lexical Items Learned 158
8.4 Generation and Incomplete Specification 160

9 Conclusions 163

Appendix

A Sample Grammar 167
  A.1 General Principles of the Grammar 168
  A.2 Rule Schemata 171
    A.2.1 Schemata for Headed Structures 171
    A.2.2 Schema for Non-Headed Structures: 179
  A.3 Lexical Rules 182
  A.4 Lexical Entries 187
    A.4.1 Definitions that Cut across Syntactic Categories 187
    A.4.2 Adjectives 188
    A.4.3 Adjectival Prepositions 190
    A.4.4 Adverbials 192
    A.4.5 Conjunctions 200
    A.4.6 Determiners 201
    A.4.7 Markers 204
    A.4.8 Nouns 205
    A.4.9 Verbs 211

B Text Corpus Used For Grammar Development 229

C Real-World Knowledge Base 244
  C.1 Slot-Filler Information 244
    C.1.1 Relation-Role Information 244
    C.1.2 Modifier-Modified Information 257
  C.2 Semantic Type Information 261

D Learning Procedure Code 267
  D.1 Starting a Learning Session 267
  D.2 Making a List of Candidates 268
  D.3 Trying the Candidates One at a Time 269
    D.3.1 Constructing a One-Word Grammar 271
    D.3.2 Instantiating Successful Lexical Hypotheses 274
  D.4 Reading and Writing Hypotheses to Files 278
  D.5 Combining New and Previous Hypothesized Definitions 279
  D.6 Utilities 286

E Sample Learning Episodes 290
  E.1 Sample Output for Each Syntactic Category 290
    E.1.1 Adjective 291
    E.1.2 Adjectival Preposition 296
    E.1.3 Adverb 305
    E.1.4 Determiner 311
    E.1.5 Noun 315
    E.1.6 Intransitive Verb 319

E.1.7 Transitive Verb 323
E.2 Converging on a Lexical Entry through Multiple Inputs 327
E.2.1 The driver glarf left 327
E.2.2 The driver glarf stopped 340
E.2.3 Merge of previous two entries 353
E.2.4 The driver stopped glarf 360
E.2.5 The vehicle stopped glarf 366

Bibliography 373

Vita 380

LIST OF FIGURES

4.1 Representation of "Fish swim" without bar-levels. 66
4.2 Representation of "Fish swim" with bar-levels. 67
4.3 An example of iterative coordination with a single conjunction. 71
4.4 An example of iterative coordination with multiple conjunctions. 73

6.1 Algorithm for constructing a version forest from a single input. 101
6.2 Lexical version space for a newly encountered determiner. 108
6.3 Abbreviated lexical version multi-space for the unknown "glarf" in "John glarf left." 110
6.4 Full lexical version multi-space for the unknown "glarf" in "John glarf left." . . . 112

7.1 Algorithm for combining lexical version multi-spaces. 125
7.2 Lexical version multi-space for "John glarf left." 132
7.3 Lexical version multi-space for "John glarf stopped." 133
7.4 Lexical version space for a new determiner. 134

8.1 Syntactic Phrase Marker for the sentence "Heading toward the interstate now would be a good idea." 150
8.2 Words of one syntactic category used as another. 152

Chapter 1

Introduction

One of the obstacles to be overcome in Natural Language Understanding is the existence of lexical gaps; that is, words or word senses which are not in the lexicon of the system. No lexicon, whether hand-coded or derived from an on-line dictionary, can ever be complete, in the sense of having entries for every word encountered in every syntactic category and with every semantic sense with which it may be used.

In order to address this issue, this thesis describes the implementation of MURRAY, a learning mechanism which is able to (i) infer the syntactic properties of a new lexical item from its syntactic environment; (ii) infer the meaning of a novel lexical item based on context and a domain-specific database of real-world knowledge; and (iii) combine those syntactic and semantic properties of a given unknown word inferred from multiple pieces of input, resulting in a version space of possible lexical entries for the unknown, each consistent with all environments in which it has been encountered.

The first section of this chapter gives an overview of the range of lexical gaps to be found in natural language data, and describes the desired behavior of a parsing system on encountering such data. Section 1.2 then gives an outline of the remainder of the dissertation.

1.1 Focus of the Research

The process by which people acquire their native language is one of the central problems of understanding the human mind. Research in this area has often focused exclusively on the type of language acquisition that occurs only in young children, but it is clearly not only children who are engaged in acquiring a native language. Adult speakers of a language regularly encounter new lexical items and syntactic constructions. In fact, [Slator, 1991] provides evidence that the rate at which new lexical items are encountered remains fairly constant over time.

But adults have the advantage of having a large repertoire of known lexical items, creating a framework within which to understand new words. Adult knowledge also includes syntactic constructions to use in creating a linguistic context, and a large corpus of knowledge about the world to provide contextual background knowledge for inferring the meanings of new lexical items. The use of an existing linguistic system and a background knowledge-base in learning has seldom been explored in the study of language acquisition. The learning system described here makes extensive use of such knowledge, and may therefore be thought of as analogous to adult learners acquiring new words of their native language.[1]

1.1.1 The Goal: Understanding All Lexical Items Encountered

I am assuming that one goal for a Natural Language Understanding system which strives for the best possible performance is that it should either have in its lexicon, or be able to infer the usage of, all of the lexical items it encounters. However, this has not been a goal for all Natural Language Processing (NLP) systems. In the early days of NLP, Weisenbaum's ELIZA [Weisenbaum, 1966] gave the appearance of understanding by responding to a few key words in each input utterance. A similar, though more sophisticated approach is taken by Schank and his followers (*e.g.* [Schank, 1975], [Schank and Abelson, 1977]). These systems also perform an analysis based solely on 'content' words, ignoring syntax (except for surface word order) and function words, such as determiners and prepositions.

More recently, systems such as General Electric's NLToolset ([Jacobs and Rau, 1988]) have incorporated methods of constructing partial analyses when it is not possible for the system to process every word of the input. A complete syntactic analysis is attempted, but words which are not known are simply ignored. The resulting analyzed fragments are given as input to a module that infers a likely interpretation, using world knowledge.

---

[1] The system could also have applications for an NLP system that operated in one language and was being taught another. In this case, the model of the adult second language learner would be more appropriate. I do not explore this possibility here.

However, there are inherent limitations to the degree of understanding that such systems can achieve. Although they can get a good percentage of understanding with less computational effort by employing certain heuristics to make up for less than complete lexical knowledge, reliance on such heuristics brings its own set of dangers. For example, a system based on Schank's Conceptual Dependencies can easily be fooled by a sentence describing an unusual state of affairs. Such systems confidently interpret both (1.1) and (1.2) as describing a situation where a dog bit a man.

(1.1) The dog bit the man.

(1.2) The dog got bit by the man.

This is partly due to the reliance on surface order of content words, while ignoring function words and syntactic constructions, and partly to the strong reliance on a knowledge base, which specifies which entities in the world are more likely to bite which other entities. Sentences that violate these expectations are likely to be misunderstood. However, since one of the most important functions of natural language is to communicate new information, sentences conveying unexpected facts are precisely those which it is most important to understand.

It is certainly the case that, all other things being equal, a system which did understand every word of the input would be superior to one which did not. Of course, in the real world, all other things are not equal. Designers of systems such as the NLToolset have deliberately chosen to work with partial analyses because this is computationally cheaper and gives reasonably good results. In a system that must deal with large amounts of data in a short time, less than complete understanding is an acceptable trade-off for efficiency. However, I wish to explore what would be required to achieve complete understanding of a text, with efficiency as a secondary consideration.

1.1.2 Lexical Gaps in Natural Language Text

There are a number of reasons that incompleteness in unavoidable with any lexicon. First, of course, is the issue of words which are simply not known. Even the most ambitious on-line dictionary, one which combined the resources of numerous published dictionaries and other resources, would not contain every word that is encountered in naturally occurring text. This is because new words are continually entering the language, and may be coined by a writer or speaker at any time. In addition, many specialized fields contain technical terms which are not likely to be found in standard reference works of a language. New words are the primary focus of this thesis. The desired behavior here is to learn the word; that is, to infer its relevant properties and add an entry for it into the lexicon.

For other types of lexical gaps, there may be different desired behavior for an NLP system. It is impossible to enummerate all possible proper nouns that occur in natural language text, so these are a continuing source of lexical gaps. However, it is not clear that all proper nouns should be learned, in the same way that new common nouns should be. Assuming that memory storage is finite, it may become prohibitively expensive to learn and store all proper names encountered. The desired behavior in this case would be to recognize a proper name as such, infer its relevant properties for the given context, but not store it. I do not here address the issue of how to recognize that an unknown word is a proper name, and so they are here treated the same way as other unknown words.

At a more practical level, simple misspellings may not be recognized as such by the system. The desired behavior for the system in this case would be not only to recognize the unknown as a misspelling, but to infer what the correct word should have been. Depending on the purposes of the system, it may be desirable to add the misspelling to some type of database, so that a similar misspelling can be identified quickly in the future. However, the misspelling should not be added to the lexicon in a way that would allow it to be used in generation.

Guo describes a procedure for recognizing occurrences of metaphor and misspellings [Guo, 1987], so that by the process of elimination, the need for learning new word senses can be detected. Guo distinguishes two kinds of misspellings. Context independent ones are those which do not conform to the pattern of any known word. These are caught in preprocessing and submitted to the user for correction. In context dependent misspellings, the misspelled word is the same as another, correct word in the lexicon. This causes a violation of semantic preferences. Since the reasoning involved in recognizing misspellings falls under the category of error analysis rather than learning, it is outside the scope of this thesis. I therefore treat misspellings as unknown words.

Another source of lexical gaps is the fact that words of one syntactic category can productively be used in another syntactic category. This is illustrated by the examples in (1.3).

(1.3) a. This book is a great read.

b. First you hit the delete button.

c. This phenomenon is not surface observable.

d. The *Capriccio* is written for a tuba player who can circular breathe.

e. The youths were wilding in the park.

Even within a syntactic category, a word can have multiple senses, as discussed, for example, in [Zernik, 1991] and [Slator, 1991]. I discuss these phenomena in Chapter Eight, and sketch a treatment of them involving the lexical rules of Head-Driven Phrase Structure Grammar, but this treatment has not been implemented into the parser and learning mechanism.

Similarly, a given sense can be used to refer to things other than its literal denotatum in contextually-dependent ways. This is discussed in [Nunberg, 1978], with examples similar to (1.4).

(1.4) a. The ham sandwich at Table 6 would like some coffee.

b. I talked to the power company on the phone, and they'll be here tomorrow.

c. The Beckman Institute is going Christmas caroling next week.

Finally, idioms and metaphorical usages prevent a linguistic element from being correctly understood under its literal interpretation. A metaphorical usage comes from relaxing the semantic preference of a predicate. Here as well, I provide a sketch of a treatment for these phenomena in Chapter Eight, but they are outside of the scope of the implemented system.

1.1.3 The Material to be Learned

In learning a new word, there are several major kinds of information that the system should infer for it. These include its phonology and/or orthography, syntax, semantics, pragmatics and stylistics. In this dissertation, I focus primarily on syntax and semantics. Syntactic information includes not only the part of speech of a word, but also, among other things, the number and type of arguments it requires, if any, the sort of element it modifies, if it is a modifier, and factors which influence its morphological form, such as number, tense and aspect.

"Semantics" is a broad term, and has been used in many different ways in the literature of linguistics, philosophy and computer science. In general, I use the term to refer to what is known as truth-conditional semantics. I am assuming a framework of compositional semantics, in which the semantic properties of linguistic constituents are combined by the grammatical rules of the language, to determine the semantic properties of the sentence as a whole. The properties of the sentence are then compared with a model of the world, from which the sentence can be labeled as true or false.

The parsing system used in this work deals exclusively with written text, rather than spoken language. As is the practice in Head-Driven Phrase Structure Grammar, I sometimes refer to the `PHON` feature, which is mnemonic with 'phonology,' but in practice, this feature is used to represent surface orthographic forms. The learning mechanism should, and does, acquire orthographic information for every new word learned. However, as will be seen in the description of the algorithm, this is trivially easy, and falls out of the algorithm without having to say anything special about it. This is due to the fact that in the grammatical formalism adopted here, orthographic (or phonological) information is represented in the same way as other kinds of lexical information.

1.2 Outline of the Thesis

The issue of lexical gaps is a large and multi-faceted problem, and there are many possible approaches to dealing with it. Chapter Two is a comparison of my research with existing work, along the dimensions of the initial given lexicon and the amount of interaction with the human user. In Chapter Three, I give the theoretical background necessary for an understanding of the learning mechanism, including the grammatical framework and the parsing strategies employed. Chapter Four describes the components of the system with which the learning mechanism interacts, including the grammar and the database of domain-specific real-world knowledge. Chapter Five proposes a modification to the feature structure system of [Pollard and Sag, In press], motivated in part by efficiency concerns in connecting the parser with the database of real-world knowledge.

Chapters Six and Seven describe the algorithm for inferring the syntax and semantics of new lexical items. Chapter Six tells how, on a single encounter with an unknown word, the system constructs a space of hypotheses about its feature structure. Chapter Seven describes the procedure for combining the hypotheses generated from a single encounter with an unknown with the hypothesis set constructed from previous encounters with that item. These are combined to form an updated hypothesis set. Chapter Eight discusses issues which have not been implemented in the learning mechanism. Some of these are due to memory constraints, while others would require a reworking of the grammar processing system itself.

Finally, there are appendices for significant pieces of the code used in implementing the learning procedure. These include a sample grammar, a portion of the world knowledge base, the code of the learning procedure, and sample learning episodes containing the input and output.

Chapter 2

Relation to Existing Work

In this chapter, I discuss two dimensions of lexical acquisition and compare the positions I take along these dimensions with those of existing research. First, I discuss the dimension of the initial state of the lexicon, and then that of the method of learning involved. In each section, I describe several positions which have been taken along the given dimension, and provide brief overviews of two or three representatives of each position. Following these overviews, I discuss the issues raised with each position, and compare and contrast it with the approach I am taking.

2.1 Initial State of the Lexicon

In investigating issues of lexical learning, there are a number of initial states of the lexicon that one could take as a starting point. One could assume no prior lexical knowledge, and build the lexicon from scratch. On the other extreme, one could assume what is supposed to be a completely functional NLP system, and look at areas where it is found to be incomplete. It is also possible to take an intermediate stance, starting with a partial lexicon and seeking to extend it.

In this section, I describe representatives of each of these positions and the differing goals and assumptions that underlie them. I then offer a justification for the position I am taking along this continuum, which is that of assuming a relatively complete lexicon to begin with. I argue that while research under any of these sets of assumptions may be profitable in different ways, extending a complete lexicon is the most likely to produce a benefit for existing NLP systems, dealing with languages where the structure of the lexicon is largely known.

2.1.1 Building the Lexicon from Scratch

One possible focus of research on lexical acquisition is on building a lexicon from scratch, with no prior information about particular lexical items, or the overall structure of the lexicon. This is generally done in the context of a system which is at the same time building syntactic knowledge from scratch. The goal of such research is not simply a system which mimics the abilities of a human language user, but one which emulates the linguistic development of a child. Positive results in such research are taken to be not only correct output, but also an order of acquisition which correlates with that empirically observed in human children, and errors of kind and frequency similar to the errors of children.

I here describe two representatives of this approach to lexical learning, and then discuss some of its limitations. I then discuss the role of human-like errors in a learning system, as it is viewed in this approach as contrasted with the approach of this thesis.

2.1.1.1 Siskind's MAIMRA System

One example of a system which builds a lexicon from scratch is Siskind's MAIMRA system [Siskind, 1990], which focuses on learning single words, based on extra-linguist stimuli. The goal of the system is to acquire a lexicon with no prior knowledge of the lexical categories or meanings of any of the words encountered. "Learning" a word here means acquiring its syntactic category and a link between the word and the real-world concept that it denotes.

As input, the system receives correlated streams of linguistic and non-linguistic information. The non-linguistic information consists of a scenario, which is an ordered sequence of scenes, while the linguistic information consists of an ordered sequence of sentences. The output is a disjunctive set of possible lexicons, each consistent with the constraints imposed by the input so far.

The system has three components, the Parser, the Linker, and the Inference Component. It is claimed that the Context-Free Grammar used by the Parser, though admittedly incomplete, is part of an innate grammar, based on claims of X-bar theory. This claim seems somewhat suspect, and it is not defended in detail, but the validity of the system does not depend upon it. The grammar, given in (2.1) is highly ambiguous, and overgenerates a large number of ungrammatical sentences, but constraints in the Inference Component limit the number of

allowable sentences. The Linker contains one linking rule, also claimed to be innate, which enforces a compositional view of semantics.

(2.1) $\mathrm{S} \rightarrow \mathrm{NP}\ \mathrm{VP}$

$\bar{\mathrm{S}} \rightarrow \{\ \mathrm{COMP}\ \}\ \mathrm{S}$

$\mathrm{NP} \rightarrow \{\ \mathrm{DET}\ \}\ \mathrm{N}\ \{\ \bar{\mathrm{S}} \mid \mathrm{NP} \mid \mathrm{VP} \mid \mathrm{PP}\ \}^{*}$

$\mathrm{VP} \rightarrow \{\ \mathrm{AUX}\ \}\ \mathrm{V}\ \{\ \bar{\mathrm{S}} \mid \mathrm{NP} \mid \mathrm{VP} \mid \mathrm{PP}\ \}^{*}$

$\mathrm{PP} \rightarrow \mathrm{P}\ \{\ \bar{\mathrm{S}} \mid \mathrm{NP} \mid \mathrm{VP} \mid \mathrm{PP}\ \}^{*}$

$\mathrm{AUX} \rightarrow \{\ \mathrm{DO} \mid \mathrm{BE} \mid \{\ \mathrm{MODAL} \mid \mathrm{TO} \mid \{\ \{\ \mathrm{MODAL} \mid \mathrm{TO}\ \}\ \}\ \mathrm{HAVE}\ \}\ \{\ \mathrm{BE}\ \}\ \}$

The assumption underlying the Inference Component is that the linguistic input must explain the non-linguistic input. This entails that the human teacher who gives the input cannot say anything that is false or irrelevant to the situation. Inference rules are also innate, as justified by the fact that they are low-level and non-controversial.

The scenes making up scenarios are analysed to compute all possible events and states described. The process of fracturing, which is the inverse of linking, yields a disjunctive lexicon. There is currently a constraint of monosemy, which requires that each lexical item can have only one meaning.

Siskind argues that acquisition is a simultaneous satisfaction of a number of constraints. For example, the system is given the input sentences in the (a) examples below, and the corresponding scenarios encoded in the logical formulas of the (b) examples.

(2.2) a. The cup slid from John to Mary.

b. $(\mathrm{BE}(\mathrm{cup}, \mathrm{AT}(\mathrm{John})) \wedge \neg\, \mathrm{BE}(\mathrm{cup}, \mathrm{AT}(\mathrm{Mary})))$;

$(\mathrm{BE}(\mathrm{cup}, \mathrm{AT}(\mathrm{Mary})) \wedge \neg\, \mathrm{BE}(\mathrm{cup}, \mathrm{AT}(\mathrm{John})))$;

(2.3) a. The cup slid from Mary to Bill.

b. $(\mathrm{BE}(\mathrm{cup}, \mathrm{AT}(\mathrm{Mary})) \wedge \neg\, \mathrm{BE}(\mathrm{cup}, \mathrm{AT}(\mathrm{Bill})))$;

$(\mathrm{BE}(\mathrm{cup}, \mathrm{AT}(\mathrm{Bill})) \wedge \neg\, \mathrm{BE}(\mathrm{cup}, \mathrm{AT}(\mathrm{Mary})))$;

From this, making use of the grammar in (2.1), the system is able to acquire the following lexicon from these two correlated pieces of input.

(2.4) *The* : DET

cup :	N	$[_{\text{Thing}}\ \text{cup}\ ]$
slid :	V	$[_{\text{Event}}\ \text{GO}(x, [_{\text{Path}}\ y, z\ ])]$
from :	P	$[_{\text{Path}}\ \text{FROM}\ ([_{\text{Place}}\ \text{AT}(x)])]$
to :	P	$[_{\text{Path}}\ \text{TO}\ ([_{\text{Place}}\ \text{AT}(x)])]$
John :	N	$[_{\text{Thing}}\ \text{John}\ ]$
Mary :	N	$[_{\text{Thing}}\ \text{Mary}\ ]$
Bill :	N	$[_{\text{Thing}}\ \text{Bill}\ ]$

The system can be used in any of three directions. Given any two of Sentences, Lexicon and Observations, it can deduce the third, for generation, acquisition, and understanding, respectively.

While it claims to be building a lexicon from scratch, it is clear that Siskind's system requires a large amount of knowledge to be present at the outset. Not only the context-free grammar, but also the inventory of possible syntactic categories and semantic types (Thing, Event, Path, etc.) must be part of the knowledge of the system in order for it to acquire new lexical items. As it is not realistic to posit a learner that knows everything about a language but nothing about the words in it, appeals are made to innateness to justify the knowledge that is given.

2.1.1.2 Nicholl's CAM System

Nicholl's CAM system [Nicholl, 1991] is constructed in a way that partially addresses some of these difficulties, although it still requires a certain amount of knowledge to be hard-wired at the outset. CAM, which stands for *Categories, Agreement, and Morphology*, acquires all of these types of information at once, while also building up a PS grammar of the input language.

In contrast to Siskind, what is assumed to be innate here is not a complete grammar, but a set of constraints on what the rules of a grammar could be. These are referred to as schemata, and are given in (2.5).

(2.5) $\text{S} = (\text{NP} \mid \text{AUX})^{*}\ \text{V}\ (\text{NP} \mid \text{AUX})^{*}$

$\text{NP} = \text{Det}^{*}\ \text{N}\ \text{Det}^{*}$

$\text{N} = ((\text{Noun-Morph})^{*}\ \text{Noun-Stem}\ (\text{Noun-Morph})^{*})$

$\text{V} = ((\text{Verb-Morph})^{*}\ \text{Verb-Stem}\ (\text{Verb-Morph})^{*})$

BEST COPY AVAILABLE

As the system acquires the syntactic and morphological properties of new lexical items, it also determines the particular instantiations of these schemata that constitute the PS rules of the language. This in turn facilitates parsing and classification of words in subsequent output, yielding a bootstrapping effect.

As in Siskind's system, although the category of a new word is not known, the finite list of possible categories must be provided to the system. Similarly, the acquisition of agreement properties depends on constraints on possible patterns of agreement, which are argued to be universal, and therefore innate. The issue of semantic types is not relevant here, as Nicholl's system does not acquire links with real-world concepts.

2.1.1.3 Discussion: Limitations and Errors in Acquisition

As these systems show, there are limitations to the approach of building the lexicon from scratch. First, it is necessary to make large assumptions about what knowledge exists in the system before learning takes place. These assumptions include a complete inventory of the available syntactic categories, and some degree of specification about the grammar of the learner. For human language learners, these assumptions are, at the very least, not uncontroversial, calling into question how closely the system mirrors acquisition in children.

Moreover, if the model of child acquisition is to be taken seriously, then lexical acquisition should take place alongside of the acquisition of knowledge about the world. In Siskind's system, it is assumed that all of the perceptually salient aspects of a real-world situation have been abstracted out and systematised for the learner's attention. While it is reasonable to expect salience of particular features of the world to be the basis of acquiring word meanings, it is not reasonable to assume that all cognitive abilities of reasoning about the world are complete before word learning begins.

It is perhaps significant that none of those systems which purport to emulate a child's early lexical acquisition has been demonstrated to be able to reach an adult level of linguistic competence. This is true of the statistical and induction learning system of [Kasman, 1991], as well as [Siskind, 1990] and [Nicholl, 1991]. The difficulty that these systems have in scaling up seems to be due to the ever increasing amount of syntactic and/or extra-linguistic knowledge that is required in order to learn words with more complex syntactic and semantic structures.

In systems which claim to emulate the child's process of lexical acquisition, much is made of the occurrence of errors which are similar to those that children are observed to make, under the assumption that these errors show that the same learning mechanism is at work. In Chapters Six and Seven, I also note places in which my learning algorithm makes the same errors that people do, given similar input. However, I do not claim that this necessarily indicates that people have internalised exactly the same algorithm for lexical acquisition that I am presenting here. Rather, the similarity of errors follows from the fact that the input data itself may be misleading or difficult, and may falsely point to certain incorrect hypotheses about the lexicon, regardless of the precise mechanism by which these hypotheses are arrived at.

In contrast to the authors of these other systems, therefore, I would view a system which avoided the mistakes that people make as a positive thing, even if it indicated a divergence from the cognitive processes that people actually use in lexical acquisition. However, it may often be the case that as people take a reasonable approach to learning new information, a system which is designed to take a reasonable approach will operate similarly, and will therefore be fallible in just the same ways.

2.1.2 Extending a Core Lexicon

There are many possible initial states for the lexicon of a learning system intermediate between being completely empty and containing all lexical information available. The most useful intermediate position has been to start with a domain-independent 'core' lexicon, and then perform learning only for domain-specific lexical items. This has been perhaps the most widely investigated approach to lexical acquisition.

This option is generally discussed in relation to the issue of portability, the ease with which a core NLP system can be customised for a new domain. Clearly, a system is the most useful in a given domain if it contains all the information it needs to operate in that domain, and none of the information that is inapplicable to that domain, as such irrelevant information could interfere with that which is relevant. Therefore, in order for a system to be easily applied to a number of different domains, it should contain all and only the information that is common to all domains; *i.e.*, is domain-independent. Domain-dependent information should be learned in the context of the particular domain.

2.1.2.1 Carter's Core Language Engine

Carter's Core Language Engine (CLE) [Carter, 1989] translates sentences of English into a representation of their literal meaning. CLE has wide syntactic coverage, and is meant to be flexible enough to be used with a variety of different types of databases. VEX, for Vocabulary EXpander, is the acquisition component of CLE. VEX is designed to allow the creation of lexical items by users with knowledge of the domain, but not of linguistics or the technical workings of the system.

There are three types of information found in lexical entries. **Syntactic entries** give complex syntactic categories, using features for unification in parsing. **Word sense entries** give both syntactic and semantic information used for interpretation. **Sortal entries** give selectional restrictions.

The CLE maintains its lexical information in a hierarchical structure referred to as the **paradigm lexicon**. A paradigm contains a set of patterns, such as word order and syntactic transformations that its members may undergo. A paradigm is thus a minimal non-empty intersection of lexical entries, and every lexical entry is a disjoint union of paradigms. This is similar to the approach to the lexicon of HPSG, as will be seen in Chapter Three. In the hierarchical approach to the lexicon, acquisition consists of determining the proper place of a new item in the hierarchy, rather than building a new lexical item one feature at a time.

Close-class items are entered by hand as part of the initialization process, and are thus part of the core language, along with some general open-class items. In learning new words, VEX asks the user for grammaticality judgments and selectional restrictions. The system has been used not only in applications of CLE to particular domains, but to construct a 1200-word core lexicon, suggesting that it is useful for general-purpose as well as domain-specific lexical acquisition.

2.1.2.2 Ballard and Stumberger's TELI

Ballard and Stumberger observe the problem that existing systems with restricted input generally do not tell the user what the restrictions are. In addition, it is not possible to predict all the words and syntactic constructions that will be needed in a given domain. Their solution is to allow customization for a given domain dynamically, by the user.

TELI, the Transportable English-Language Interface System, [Ballard and Stumberger, 1986],interfaces with a relational database, and uses case frames with dynamically alterable selectional restrictions. The user supplies the vocabulary, the selectional restrictions, and definitions of words and phrases. Users can impart new concepts to the system in terms of other definitions, or as a conjunction of an arbitrary number of constraints.

In the initial phase of customization, the system asks the user to give information to interpret the database tables. This allows the system to interpret noun-noun modifications and possessive constructions, and to distinguish entities from properties. In intermediate customization, the system asks which database entries correspond to English words, and constructs definitions of those words. It tests the acceptability of syntactic and semantic constructions, and allows the user to volunteer lexical and syntactic information, such as case frames. On-line customization occurs when the system encounters undefined words, or the user wants to give a new definition. This can be done at the database level, through a menu, through English-like interaction.

The main drawback of this system is the burden that it places on the user. The user is responsible for ascertaining and maintaining the correctness of all lexical information in the system. This includes not only a great deal of linguistic knowledge, but also domain knowledge.

2.1.2.3 Discussion

With regard to acquisition, systems that have operated successfully with the core lexicon approach would apparently have been just as successful in extending a complete lexicon or building a lexicon from scratch, as was the case with Carter's CLE. The advantages of such an approach, if any, are not in its facilitation of lexical acquisition, but in providing a domain-specific context, which aids in disambiguation. The disadvantage of the core lexicon system described above is in the burden that they place on the user. This is discussed more fully in Section 2 of this chapter.

2.1.3 Extending a 'Complete' Lexicon

As was seen in the previous sections, even those systems whose purpose is to build a lexicon from scratch must start with a great deal of lexical and other kinds of knowledge, both linguistic and non-linguistic. In addition, it may be observed that a great deal of what such systems learn

2

is the kind of information that a human grammar writer would already know, and could easily encode in the system to begin with.

Therefore, a more practical alternative might be to integrate lexical acquisition into a system which already contained all the information that is needed in order to carry out NLP tasks. This has rarely been done, however, because it is not often been perceived to be necessary.

For some time in NLP, there was an unspoken assumption that on-line dictionaries could yield a complete lexicon, and that therefore learning should be unnecessary. However, as soon as this assumption was seriously examined, it was found to be false. Arguments against it are given, for example, in [Atkins and Levin, 1991], and in Chapter One of this thesis.

2.1.3.1 Trost and Buchberger's VIE-LANG

There are few examples of this approach to lexical acquisition in English. However [Trost and Buchberger, 1986] have investigated lexical acquisition for German, where new words are formed productively through compounding. They have implemented a procedure for the semi-automatic enlargement of a morphological lexicon in VIE-LANG, a German dialogue system.

VIE-LANG uses graphemic representation to classify new words according to language-specific properties of German, including inflection and case marking. New words are assigned to one of about 70 morphological classes. Lexical entries contain information about the class, as well as about umlaut, past participle formation and other forms.

Acquisition is rule-base, using inflectional, morphological, and heuristic rules. The system first determines the word type. To do this, it first checks whether the word is already in the lexicon, then perform a morphological analysis. If this does not produce a definite answer, it asks the user for a sample utterance.

Once the word type has been determined, the system activates the appropriate package of rules for that type, to do further analysis. This also involves interaction with the user, to test hypotheses along the way. The examples discussed are all particular to German, but similar principles could be applied in other languages.

2.1.3.2 Zernik's RINA

Zernik's work on the Dynamic Hierarchical Phrasal Lexicon in RINA ([Zernik, 1988] and [Zernik and Dyer, 1986]) is also of this type. RINA starts out with a lexicon that is supposedly complete for single words; the source of lexical gaps focused on here is phrases which function idiomatically. The large number of such phrases leads to a very large lexicon, so entries cannot all be made manually, but must be acquired from text.[1]

The goals of the system are to learn lexical phrases from training examples, determining the place of each phrase in a lexical hierarchy, and to use this hierarchy to assist in the learning of new lexical phrases. A *lexical phrase* is an association of syntactic pattern and conceptual meaning. Parsing consists of matching a syntactic pattern, validating the corresponding situation from the context, and instantiating the resulting concept. RINA reads a paragraph of text from the user, containing unknown words or phrases, and gives back a paraphrase representing its current understanding. In this dialogue with the user, the system first tries the literal interpretation, and asks for confirmation. If this is wrong, it forms a new hypothesis and tests that.

To create a new lexical phrase, RINA first invokes the appropriate script. The syntax of the new phrase is inferred from the sample sentence, by factoring out things that are covered by general rules. The situation of the new phrase is a generalized version of the script itself. This determines the context in which the phrase is felicitously used, which is important for disambiguation. Finally, the concept of the new phrase is extracted from the script. It is noted that selectional restrictions help in disambiguation, but this leads to problems of indirect reference, as in "Your client ran into my client" (referring to their cars).

Little is said in [Zernik, 1988] and [Zernik and Dyer, 1986] regarding the degree to which this work has been implemented. A system which fulfilled all the claims of these papers would have to be extremely large and intelligent. It seems to rely on a great deal of real-world knowledge, some of which is not just domain-specific but example-specific, and many of the examples seem contrived to give the right answers. It is also not clear whether the reported RINA-user exchanges are real transcripts. If they are, the system is either very advanced or very contrived.

If an implementation of this work can actually fulfill the promises made for it, then it is clearly an effective approach for the treatment of multi-word idioms. But while the examples

---

[1] 'Phrases which function idiomatically' in this work includes not just idioms themselves, but also collocations, such as the fact that people tend to say "strong tea" and "powerful engine," but not "powerful tea" and "strong engine."

2

and discussion are limited to idioms and collocations, the claim is made that the same approach is good and necessary for the entire lexicon. This claim requires further justification.

Both of the systems described above are designed to extend a lexicon that is in some sense complete to begin with. In this, they are similar to the learning mechanism with UNICORN that I have implemented. However, these systems both require a dialogue with the user, while MURRAY operates solely from context. This difference is discussed further in Section 2 of this chapter.

2.1.3.3 Discussion: Defining 'Completeness' of a Lexicon

I have indicated above that the learning mechanism described here is designed to operate with a lexicon that is initially complete. In this section, I now give a more detailed characterization of what it means for a lexicon to be complete. In addition, because of assumptions that are made in other approaches to the initial lexicon, I argue that the differences in the approaches described above are not as great as they might first appear.

In saying that the system starts with a complete lexicon, I mean that there is a set of assumptions which can be made in acquisition. The system can assume that it has access to a complete inventory of the syntactic categories of the language. In an HPSG grammar, this means that the complete lexical type hierarchy is known. That is, there is a complete listing of the features that the feature structure of a lexical item can have, and types of objects that can be values for each of those features.

This may be thought of as analogous to starting with with something like the lexicon of an adult native speaker of the language. This is in contrast with the systems described in Section 2.1.1 above, which claimed to be acquiring a lexicon from scratch, in analogy with what a child does. However, as I argued there, even these systems start with a great deal of knowledge about the structure of the lexicon. The only difference between my so-called complete lexicon and their so-called empty ones is that a complete one has a large number of lexical items in it already, conforming to the specified structure, while an empty has the structure, but no lexical items. As the learning mechanism makes use of information about what a possible lexical item could look like, but without making reference to existing lexical items, this amounts to no difference at all.

This entails that the system I have implemented should be equally useful for acquisition with incomplete lexicons, while those designed to build a lexicon from scratch should also be able to extend existing lexicons. The latter claim has already been seen to be true for at least the two systems of this type that we have examined. I show in Chapters Six and Seven that the former is true as well, although it is more efficient to learn one word in the context of other known words, rather than trying to acquire a large number of unknown words all at once.

A system which truly acquired a lexicon from scratch would have to acquire the inventory of syntactic categories, possible features and feature values, all at the same time. However, it is not clear that this can occur outside of general cognitive development. That is, it is at least possible that linguistic categories are formed in tandem with extra-linguistic ones. As such cognitive development is outside the scope of this work, it is reasonable to here assume a given inventory of categories, features and feature values.

Moreover, the assumption of a relatively complete initial lexicon entails a certain similarity to the core lexicon approach. With a complete initial lexicon, it is likely that a word which is unknown will pertain to some specialised domain, rather than to general world knowledge. However, in contrast to the 'core' approach, words in my approach are not distinguished as to domain-independent or domain-specific, but words may be tied to concepts which are so distinguished. While it makes sense to envision a correlation between unknown words and domain-specific terminology, I do not want to make this a hard and fast rule, so that the system is unable to learn words denoting domain-independent concepts.

2.2 Method of Learning

The second dimension along which lexical acquisition systems may be categorised is that of the method of learning which they employ. As with the initial state of the lexicon, there are two extremes, and a number of possible intermediate positions. The extremes here are that of simply learning by being told versus inferring new lexical information completely from (linguistic or non-linguistic) context, without any direct input from the user. The issues of learning become more interesting as we move toward the latter extreme.

2.2.1 Learning by Being Told

The simplest way for a system to acquire new knowledge is for a person to enter it in directly, although this hardly seems to merit the name of 'learning.' However, when it is the system, rather than the human user, that directs the course of knowledge entry, it does at least appear that the system is determining what particular information it needs to acquire. In this section, I describe several systems that elicit information from the user by means of a more or less directed dialogue.

2.2.1.1 Ballard's Layered Domain Class System

One example of learning by being told is Ballard's (1984) Layered Domain Class (LDC) system [Ballard, 1984]. The LDC consists of two major components. The Knowledge Acquisition Component acquires the vocabulary and semantics of the language being used, and determines the composition of input files. The User-Phrase Processor scans input, looking up words in a dictionary, parses the input to produce a syntactic structure, translates this into a formal query, and processes the query.

The vocabulary acquisition module, called Prep, operates in three stages. First, it asks the user to name each entity in the domain and give examples which enable it to determine the case structure of verbs taking that entity as a subject. Second, it asks the user questions to determine morphological variants, such as irregular verb forms. Finally, it acquires the semantics of adjectives, verbs and other modifiers. This is domain-specific, and must be structured according to hierarchical relationships.

The Processor uses a domain-independent Phrase Structure Grammar, and domain-dependent files obtained by Prep. The pattern file is a list of proper nouns specified for the domain. The dictionary file contains some common words initially, plus roots and inflections acquired from the user. The compact file contains case structures of verbs and pairs of entities which may be linked by prepositions. This latter allows disambiguation during parsing without the expense of a complete semantics. The macro file contains the meanings of modifiers, in a language that is directly executable by the retrieval module.

The limitation of the LDC is that it asks the same set of questions for each new lexical item, and accepts only a finite, pre-determined set of answers. The user must explicitly name every item in the domain, and provide example sentences which enable the system to determine the case structure of verbs. If the user's sentences are not properly representative, the system will fail to acquire correct information. Moreover, the system will fail to understand syntactically correct sentences describing unusual situations, since its semantics severely limits what entities can co-occur.

2.2.1.2 Lang and Hirschman's SPQR

Lang and Hirschman's SPQR (Selectional Pattern Queries and Responses) [Lang and Hirschman, 1988], is the parsing and acquisition module of PUNDIT (Prolog Understands Integrated Text). PUNDIT is a full NLP system, rather than an interface or a front end. Thus SPQR does not require carefully chosen example sentences, but learns from naturally occurring text. The knowledge acquisition is interleaved with the actual use of the system.

The portability of the system comes from the separation of domain-independent from domain-specific components. Domain-specific information is used to reject semantically anomalous parses. The system covers complete sentences as well as fragments and telegraphic utterances. Ambiguity can arise from a string being able to be analysed as either a fragment or a complete sentence. In telegraphic speech, where there are few determiners, it may also be difficult to establish NP boundaries.

The system uses a kind of bootstrapping, using available selectional restrictions to determine a correct parse, and then using the parse to learn more selectional restrictions, which are in turn used to increase efficiency in subsequent parses.

The Syntactic Analyser produces an Intermediate Syntactic Representation, which has the form of operator-argument structure. For each NP and S thus produced, the system presents the user with a head-modifier or predicate argument pattern, and queries as to its acceptability. The user is then asked to generalise, using a domain-specific `isa` hierarchy. This places quite a heavy burden on the user, who may not have correct intuitions about the selectional properties of all the nodes on the hierarchy. In addition, the system does not seem to have any tolerance for noise in the form of incorrect responses. While these judgments involve more domain knowledge than linguistic knowledge, they are still susceptible to error.

The current implementation of the system requires the user to be familiar with linguistic terminology. It is suggested that in future work, the system should phrase questions in such

32

a way that a linguistically naive user can still provide the necessary information. It is also suggested that when the system encounters a word in parsing which is not represented in the domain model, the system should suggest where the word fits in the model. However, this has not yet been implemented.

2.2.1.3 Carter's Core Language Engine

As was mentioned in Section 2.1.2.1, in VEX, the lexical acquisition component of Carter's Core Language Engine [Carter, 1989], the system must ask the user for grammaticality judgments and selectional restrictions, often relying on shaky speaker intuitions. As these may not be consistent, there is a facility for negotiating with the user.

In order to get the user to judge only syntax and not semantics, the word being acquired is presented to the user in the context of sentences in which all other content words are replaced by nonsense words. For example, in acquiring the phrasal verb "use up," the sentences of (2.6) are presented to the user for grammatical judgments (from [Carter, 1989], p. 142).

(2.6) 1 `The thingummy used up.`

2 `The thingummy used the whatsit up.`

3 `The whatsit was used up by the thingummy.`

4 `The thingummy used the boojum up very good.`

5 `The boojum was used up the whatsit by the thingummy.`

6 `The whatsit was used up for the boojum by the thingummy.`

7 `The thingummy used up existing.`

8 `The thingummy used up the whatsit that the boojum existed.`

9 `The whatsit was used up by the thingummy to exist.`

The danger of this approach is that users tend to unconsciously construct meanings for nonsense words, and then judge the acceptability of the sentences on the basis of the semantic felicity of the constructed meanings. In fact, the judgments offered by Carter for this example seem very questionable.

2.2.1.4 Barnett's Luke System

A further example of learning by being told is Luke, ([Barnett *et al.*, 1990]), the acquisition module of the Cyc system. Upon encountering a new word, the system presents the user with a number of default settings for the lexical properties of the word, which are to be confirmed or modified, and a number of grammaticality judgments which the user must make.

For example, given a newly encountered singular noun, Luke constructs a likely plural form, and asks the user to either confirm it or give an alternative. The system also presents the user with a set of sentences, with what Luke believes the grammaticality judgments are. For nouns, these seem to all focus on the issue of whether the word is a mass noun or a count noun. The following is given in [Barnett *et al.*, 1990] as the `SYNTAX` entry for the noun "city."

(2.7) `There are four cities here.` **Good** Bad

`There isn't much city here.` Good **Bad**

`We must take this to city.` Good **Bad**

`I love city.` Good **Bad**

Similar queries ask the user to classify the register of a word, using two scales. First, the word conveys the speaker's attitude toward the subject matter as elevating (*e.g.* 'counsellor'), neutral (*e.g.* 'lawyer') or deprecating (*e.g.* 'shyster'). Second, the word usage is classified as common (*e.g.* 'car'), slang (*e.g.* 'wheels'), technical (*e.g.* 'automobile') or rare/archaic (*e.g.* 'motor-car').

2.2.1.5 Discussion

While a system that elicits grammatical information or judgments from the user may be acceptable, and even helpful, for a programmer constructing an initial lexicon, it will be annoying to a user whose goal is to carry out some task by interacting with the system.

Furthermore, even for native speakers of a language, grammaticality judgments are notoriously inaccurate. Part of the problem is that the user is asked to judge the grammaticality of sentences in isolation. But for many lexical properties, such as the mass/count distinction, an appropriate context will make either sound correct. For example, "baby" is normally thought of as a count noun, and "sugar" as mass, yet the following are perfectly acceptable.

(2.8) It smells like new baby in here.

(2.9) Glucose, sucrose and fructose are the only sugars I know of.

Thus these grammaticality judgments can turn into a test of the user's imaginative ability to come up with a context in which the sentences might appropriately be used.

As a consequence of not making full use of existing linguistic and real-world knowledge, the majority of current acquisition systems place an inordinate amount of the burden on the human user. By taking advantage of existing linguistic knowledge, MURRAY is able to acquire new lexical items from context, rather than by asking the user to answer queries about the item to be acquired.

2.2.2 Learning from Statistics

This is the approach most often taken in attempting to construct lexical entries from very large text corpora, with or without human interaction. The procedure is not claimed to emulate that of a human language learner, but to achieve good results in an efficient way.

2.2.2.1 Church: Statistical Tools

Church et al. describe some of the tools currently available for studying machine-readable corpora, including concordancing, which uses the Key Word In Context approach, and indexing [Church *et al.*, 1991]. British lexicographers are working on transforming machine-readable corpora into detailed lexical syntactic descriptions. Even with the most advanced available software, this process still requires human interaction.

Two statistical methods are discussed. The first, mutual information, uses co-occurrence statistics, measuring similarity among related words. The second, the t-test, measures dissimilarity among related words. This is necessary because while the information contained in most dictionaries enables the system to identify related words, it is not useful for differentiating among similar ones. This statistical measure tells us, for example, that "strong support" is more likely than "powerful support."

2.2.2.2 Dyer: Integrating Symbolic and Connectionist Approaches

Any connectionist system is statistically based at heart, because in learning, new connections are formed with a strength proportional to the number of times that a positive result has been obtained from a similar context in the past. Dyer describes an integration of symbolic and connectionist approaches [Dyer, 1991]. According to the former, knowledge is represented as relations among symbols. There are two alternatives within the connectionist approach. In the localist view, each unit represents a primitive object, while in the distributive view, each object is represented as a pattern of activation over units. There is claimed to be a need for synthesis, because while NLP appears to require symbolic processing, the brain works connectively. Word meanings in this approach have fussy boundaries, and are acquired incrementally.

A PDP net is trained with input/output pairs, and patterns of activation come to represent a symbol. Problems include the selection of microfeatures, the lack of constituent structure, a multiplication of microfeatures only applicable to a few symbols, and the distinguishing of specific instances form general concepts. Symbol recirculation allows symbols to enter into structured relationships with each other without having microfeatures, using patterns of activation.

Symbols are maintained in a separate connectionist net, called a Global Symbol Lexicon, and are modified through their interaction with other symbols. In these methods, words with similar semantics wind up with similar distributive representations. Ambiguous words get a composite representation. The system is tested with tasks of paraphrasing and acquiring scripts, which are then used to infer missing roles and events. One limitation of this approach is that connectionist systems cannot tell the difference between logical and statistical rules.

2.2.3 Applying General Models of Learning

Research in artificial intelligence has produced a number of models of learning, and implementations which are able to acquire new concepts in a given domain. In this section, I discuss attempts to apply these general models of learning to the specific problem of natural language acquisition.

2.2.3.1 Miller and Laird: SOAR

SOAR [Laird *et al.*, 1987] is a generalised problem solver, intended to model a unified theory of cognition. Learning consists of acquiring a new chunk, which is an operationalised summary of a subgoal. By acquiring a chunk for a subgoal, the system no longer has to determine a way to solve that subgoal, but has a solution ready-made to pull out of an every-increasing inventory of such solutions.

Miller and Laird [Miller and Laird, 1991] describe a model of supervised concept learning, with the following psychological constraints. (1) Learning occurs in a unified theory of cognition. For this purpose, SOAR is the basis of the learning algorithm. (2) Learning is incremental, which is achieved by the mechanism of chunking in SOAR. (3) Learning exhibits typicality effects. Typical instances should be processed faster than marginal ones. (4) Learning should exhibit underextension errors early on. (5) Learning should be sensitive to the system's beliefs, goals and theories.

The goal of the learning task is to learn the association between an object description and a word. In SOAR, learning means the construction of a new, more specific production. New instances are learned more quickly as more are seen, and so it is more likely that a given new instances will fit the template of a known production. This shows the desired typicality effects.

However, when a large number of productions have been learned, many productions might be able to apply in a given instance. This leads to contradictions, in which case nothing is done. This is actually beneficial, as it leads to the desired underextension behavior described above.

2.2.3.2 Rayner *et al.*: EBL

Rayner et al. apply the methods of Explanation-Based Learning (EBL) in lexical acquisition [Rayner *et al.*, 1988]. The trees of successful parses are used as the explanations, to which subsequent parsing attempts refer. Given a set of example sentences and a grammar, the system can infer a lexicon. Each sentence either further instantiates or refutes the lexicon, showing the need for a learning episode.

The order in which sentences are processed is critical; the system itself must order sentences so that the minimum number of new words is presented at each stage. When the context of a given sentence allows more than one possible analysis of a new lexical item, two or more entirely separate duplicate lexicons are created. Managing this large number of duplicate lexicons is a serious drawback of this system. Ideally, further input eliminates some of these duplicate lexicons, by disambiguating the lexical entries which differentiate them.

However, rarely occurring words do not get their lexical category disambiguated, and so lead to a proliferation of lexicons. There are two possible solutions to this. The first is simply to remove difficult entries. This saves only reliable information, not that which is dubious. But it fails to take advantage of the fact that words may become disambiguated by the later disambiguation of other words. A better solution is lexicon compaction, in which lexicons share common substructures. An ambiguous entry is listed as having a disjunction of lexical categories. In this way, no information is lost, but there is still the same gain of efficiency.[2]

There are things that the system cannot learn, since it uses only positive examples. For example, it can never decide whether "belong" takes a complement prepositional phrase headed by "to", or is intransitive, and just happens to always be modified by optional "to"-phrases in the sentences encountered.

Like Rayner, I assume only positive input; the input consists solely of grammatical sentences to be understood, not ungrammatical ones to be critiqued. The resolution of ambiguities in the face of only positive data is discussed in Chapter Seven.

2.2.4 Learning from Context and Real-World Knowledge

In this section, I discuss other systems which take the same general approach to lexical acquisition as mine, learning from context and/or making use of real-world knowledge. Not surprisingly, these systems also have other similarities to my work, for example, acquiring features of the learned item by means of unification.

2.2.4.1 Jacobs and Zernik's RINA and TRUMP

Jacobs and Zernik describe a method of acquiring lexical items as a gradual process, using a sequence of example text and multiple sources of information [Jacobs and Zernik, 1988]. It is performed by cooperation between RINA and TRUMP.

---

[2] As is discussed in Chapter Three, the use of underspecification in unification-based grammars gives us the effect of lexicon compaction without any special stipulations.

The use of partial knowledge to learn from text allows an analysis of sentences even where not all words are known, and opens the way for the generation of hypotheses of word meaning. This represents a change for RINA, from interaction with the user to learning from context. An ongoing problem is how to know when multiple examples motivate separate lexical entries.

The approach described here represents a combination of methods. It finds the meaning of a modifier, given the meaning of a whole phrase and prior discourse context, using linguistic clues.

The input to TRUMP is NL text, and its output is a chart of possible syntactic and semantic interpretations. The clues it uses in constructing these include morphology, syntax, semantic/conceptual (appeals to world knowledge), semantic/lexical (using semantic of nearby words), and context. TRUMP produces a list of possible interpretations for unknown words, using a correspondence between linguistic and conceptual assumptions.

RINA takes this chart as its input, along with a given context of discourse objects. Its hypothesis creation has the following steps:

- find the scope of the unknown element in the chart, using a known reference/referent association
- convert the element to a general template, by the process of variabilization
- form lexical hypotheses from given assumptions
- refine the hypotheses with further examples.

It is not clear from the description given in [Jacobs and Zernik, 1988] how much of this work has been implemented. If it has been implemented, it represents the integration of a number of modules which are large and complex in themselves. Otherwise, it provides a sketch for the shape of work to be done in the future, as these other components are implemented.

2.2.4.2 Lytinen's LINK System

The work of Lytinen and others on the Link system [Hastings *et al.*, 1991] takes an approach which is very similar to that which I am arguing for here. Like the system described in this thesis, LINK operates in the framework of HPSG, with unification as the primary mechanism of lexical acquisition, and learns over several inputs of a word in context. It is designed for a particular domain, that of an automobile assembly line, with domain knowledge constraining semantic representation.

As with Unicorn, syntax and semantics have the same representation. Violation of semantic constraints leads to a failure to parse, hence a learning opportunity. On seeing an undefined word, the system enters a link for it, with an unspecified category. It is then given whatever category fits the available rules. The system does not retain all sentences with a given word, but just the current hypothesis set and the most recent input. Clearly, there must be cases in which there is more than one initially possible category, but these are not discussed.

The system is given a complete inheritance hierarchy of semantic predicates. Here, the unification of two nodes that are in an ISA relation results in the more specific of the two. The inheritance hierarchy is used to refine the initial hypothesis. It is assumed that each possible word meaning corresponds to an existing node in the hierarchy. Therefore, the techniques of Mitchell's version spaces [Mitchell, 1977] can be used for hypothesis refinement. My own use of something like Mitchell's candidate elimination algorithm is discussed in Chapter Seven.

One limitation of the system is that it is able learn ambiguous items only if each sense has a distinct syntactic category. For example, the ambiguity of "watch" between the senses of 'look at' vs. 'timepiece' is not a problem, as these are of separate syntactic categories. However, the two senses of "star" either 'celestial object' or 'celebrity' cannot be learned, as they are both nouns.

Much of the difference between the Unicorn learning mechanism and that of Lytinen in the areas on which the two concentrate. In seeking to deal with ungrammatical utterances and metaphor, [Lytinen, 1992] abandons compositional semantics in favor of a complete reliance on semantic nets. I do not address the issue of ungrammatical utterances, but discuss in Chapter Eight a treatment of metaphor which preserves the compositionality of semantics.

Chapter 3

Theoretical Background

The learning mechanism described in Chapters Six and Seven operates in the context of an existing grammar, grammar processor, and world knowledge base. The grammar processor is the Unicorn system, developed by Dale Gerdemann and Erhard Hinrichs, while the grammar and knowledge base were written in conjunction with the present research. This chapter provides the background necessary for understanding the grammar and knowledge base, which are introduced in the following chapter, and describes the grammar processing system.

Section 1 introduces the grammatical theory used here, starting with context-free phrase structure grammars, and then describing the insights of Generalised Phrase Structure Grammar and Head-Driven Phrase Structure Grammar. Section 2 provides a brief overview of Earley's algorithm, Shieber's extension to it, and the optimisations which Gerdemann implemented to create Unicorn [Gerdemann, 1991]. Section 3 discusses advantages of unification in grammar processing, and the applications of context-free methods of parsing with unification-based grammars.

3.1 The Grammatical Theory Underlying HPSG

The purpose of this section is to build up an understanding of the grammatical framework within which the learning mechanism operates, Head-Driven Phrase Structure Grammar (HPSG). I assume a basic familiarity with context-free Phrase Structure Grammars (CF-PSGs), and start with a discussion of why context-free grammars have been thought not to be a practical means of accounting for the phenomena of natural language. This impracticality motivates the notion of Generalized Phrase Structure Grammar (GPSG), employing unification in the PS rules. I then present the advancements to the theory contributed by HPSG. The description of HPSG given here is meant to cover only those aspects of the theory that are relevant to the learning mechanism. Fuller descriptions are found in [Pollard and Sag, 1987] and [Pollard and Sag, In press].

3.1.1 Context-Free PSGs

HPSG is a type of Context-Free Phrase Structure Grammar (PSG), and thus consists of (i) a set of nonterminals, (ii) a set of terminals, the lexical items of the language, (iii) a set of context-free Phrase Structure rules, and (iv) a nonterminal designated to be the start symbol, usually *S*. Context-Free Phrase Structure rules are production rules consisting of a single nonterminal on the left-hand side (lhs) and one or more nonterminals and/or terminals on the right-hand side (rhs). In most recent linguistic theories, including GPSG and HPSG, the rhs must consist of either a single terminal, or one or more nonterminals, but not a combination of terminals and nonterminals.

The nonterminals of CF-PSG have generally been assumed to be atomic; that is, to have no internal structure. However, such CF-PSGs have been argued to be inappropriate for generating all and only the sentences of any natural language. Languages contain dependencies which may be realized on constituents that occur at great distances from each other in surface form. One well-known example is subject-verb agreement. An adequate grammar of English must generate sentences (3.1) and (3.2), but not (3.3) and (3.4).

(3.1) The car that I saw abandoned in the parking lot was old and rusty.

(3.2) The cars that I saw abandoned in the parking lost were old and rusty.

(3.3) * The car that I saw abandoned in the parking lot were old and rusty.

(3.4) * The cars that I saw abandoned in the parking lost was old and rusty.

However, the dependent elements in these sentences, the singular or plural forms of the noun "car" and the verb "be," are not next to each other. In fact, a subject and the verb which agrees with it may be separated by an arbitrary amount of intervening material.

Facts such as these have been used (for example, by [Bach, 1974]) to argue that English and all other natural languages are not context free languages. This argument is not valid, however.

Sentences (3.1)–(3.4) merely show that the single PS rule (3.5) is not sufficient to account for the facts of subject-verb agreement. Rather, the two rules (3.6) and (3.7) are required, where the subscripted categories are all still atomic.

(3.5) $\mathrm{S} \rightarrow \mathrm{NP}\ \mathrm{VP}$

(3.6) $\mathrm{S} \rightarrow \mathrm{NP}_{\mathrm{sing}}\ \mathrm{VP}_{\mathrm{sing}}$

(3.7) $\mathrm{S} \rightarrow \mathrm{NP}_{\mathrm{plu}}\ \mathrm{VP}_{\mathrm{plu}}$

The difficulty here is not theoretical, but practical. For each feature involved in any dependency, a new subscript must be introduced. The inventory of nonterminals must then include an atomic category for each combination of subscripts that can occur on any given syntactic category. This number quickly becomes unmanageably large, so that it becomes infeasible to list all the nonterminals of the grammar.

The solution is to abandon the notion that nonterminals are atomic. Instead, the information that was encoded as subscripts in (3.6) and (3.7) is recorded as values for a particular feature. These rules would then be written as in (3.8) and (3.9).

(3.8) $\mathrm{S} \rightarrow \mathrm{NP}_{[\mathrm{NUM\ sing}]}\ \mathrm{VP}_{[\mathrm{NUM\ sing}]}$

(3.9) $\mathrm{S} \rightarrow \mathrm{NP}_{[\mathrm{NUM\ plu}]}\ \mathrm{VP}_{[\mathrm{NUM\ plu}]}$

The subscripted categories here, in contrast to those in (3.6) and (3.7), are not atomic categories with complicated names. Rather, they are complex categories with internal structure. Each subscript is a **feature-value pair**, where the first element is some parameter for which the category has a value, and the second is the value for that feature found on this particular instance of the category.

While this allows us to represent a potentially infinite number of nonterminals in a CFG without actually listing them, it still fails to capture the fact that the number features on a subject and a verb phrase must be the same. In these rules, the number features of both daughters are stipulated, and only observation of the rules tells us that they are the same. In order to capture this fact, rules must allow one element to refer to structure found in another element. One way of showing this is by the use of variables. The above two rules could thus be collapsed as follows, where α is a variable ranging over possible values for the feature NUM.

(3.10) $\mathrm{S} \rightarrow \mathrm{NP}_{[\mathrm{NUM}\ \alpha]}\ \mathrm{VP}_{[\mathrm{NUM}\ \alpha]}$

Taking this one step further, it is not necessary to ascribe any special status to the syntactic category information in PS rules. Instead, the category may be represented as a set of feature-value pairs, just like any other information about a constituent. Each element in a PS rule is now seen as a bundle of feature-value pairs. On this view, (3.10) is merely a convenient abbreviation for (3.11), where CAT stands for 'syntactic category.'

(3.11)
$$[\text{CAT S}] \rightarrow \begin{bmatrix}\text{CAT NP}\\ \text{NUM } \alpha\end{bmatrix} \begin{bmatrix}\text{CAT VP}\\ \text{NUM } \alpha\end{bmatrix}$$

In fact, it may be observed that the symbols 'NP' and 'VP' themselves encode two different pieces of information, and being able to deal with them separately may facilitate the statement of grammatical generalisations. First, they encode the syntactic category itself, what is traditionally called the part of speech. This is what 'NP' and 'N' have in common, but do not share with 'VP.' In GPSG and HPSG, the term 'category' is used to refer to a larger subset of the syntactic features of a constituent, and this notion of the part of speech is called the 'major category.'

Secondly, there is the notion of being phrasal, which 'NP' and 'VP' share, in contrast to 'N' and 'V.' There is at least a two-way distinction, between lexical nodes, called preterminals, which immediately dominate lexical items, and other nonterminals, which are called phrasal. Separating out this information into distinct feature-value pairs, (3.11) now becomes (3.12), where [LEX 0] is used to denote a phrasal category.

(3.12)
$$[\text{CAT S}] \rightarrow \begin{bmatrix}\text{CAT N}\\ \text{LEX 0}\\ \text{NUM } \alpha\end{bmatrix} \begin{bmatrix}\text{CAT V}\\ \text{LEX 0}\\ \text{NUM } \alpha\end{bmatrix}$$

In many theories of syntax, the phrasal-lexical distinction has more than two possible values. Jackendoff popularised a system called X-bar syntax [Jackendoff, 1977], in which this parameter is referred to as the bar-level of the category. Numerous researchers have taken Jackendoff's work as a starting point,[1] generally using either two or three bar levels, in addition to bar-level zero, which denotes preterminal nodes.

---

[1] For discussion, see [Pullum, 1985]. The theoretical content of this paper is superseded by [Kornai and Pullum, 1990], but the latter lacks the historical overview of the often amusing fashion in which the notion of X-bar theory has been invoked in the literature, without any specification of the particulars of the theory.

3.1.2 Gazdar's GPSG

A PSG which allows nonterminals to be complex feature structures rather than atomic categories is referred to as a **Generalised Phrase Structure Grammar.** This is also the name of a theory of syntax that uses complex nonterminals, described fully in Gazdar, Klein, Pullum and Sag [Gazdar *et al.*, 1985a]. They posit a system of hierarchical relationships among features and values, so that the value of a feature may itself be a set of one or more feature-value pairs.

Gazdar et al. also elucidate principles for stating dependencies among the feature values of different elements of generalised PS rules. According to these principles, the co-varying feature values in a rule do not merely have the same form, rather they are the same entity. That is, they are **token-identical,** not just **type-identical.** For example, in rule (3.12) above, the two occurrences of α are not two identical specifications for the relevant feature values, but two references to the exact same feature value.

Parts of an element's feature structure may be lexically unspecified. Some unspecified values may be filled in by virtue of dependencies with other elements. For example, the English noun "fish" is unspecified for number. However, "is" is specified to take a singular subject, so we know that "fish" is singular in (3.13).

(3.13) The fish is swimming upstream.

On the other hand, modal auxiliaries such as "ought" are unspecified for number for the number of their subject, while most nouns have have a number specification, as "duck" does. So in (3.14), the number specification on the verb is determined by its subject.

(3.14) The duck ought to swim upstream.

Of course, an unspecified subject may combine with an unspecified verb, as in (3.15), in which the subject may denote one fish or many fish.

(3.15) The fish ought to swim upstream.

However, any well-formed utterance of a language must be fully specified. That is, it must have a value for every feature which is appropriate to its type. Therefore, rather than saying that sentence (3.15) is unspecified for number, we say that it may be specified as either singular or plural. Therefore, this sentence is not vague as to its number, but ambiguous between two fully specified forms.

It is common for two elements of a dependency to both be specified for a certain feature-value, as in (3.16).

(3.16) The duck is swimming upstream.

It is incorrect to say that the NUM feature in such examples is inherent on either the subject or the verb and moves, or gets copied to the other. Rather, the subject and verb each have specifications (including possibly null specifications) for these features. The PS rule which licenses the subject and verb as daughters of a local subtree requires that the feature specifications be token-identical. Therefore, even if each specifies only a subset of these feature values, the specifications must be consistent.[2]

Two feature values are consistent if and only if they have a non-null unification. As the two values are structure-shared, they each have the value that is the unification of those values which occur on them by virtue of specifications at ay lower level of the tree.[3] For example, while the lexical entry for "fish" is unspecified for number, the token of "fish" in sentence (3.13) has the specification [NUM sing], because of its dependency with the verb, which is lexically specified as taking a subject that has the feature-value [NUM sing].

With a hierarchical feature structure, dependencies may be stated in a more general way. It is not necessary to stipulate that in English, subject and verbs agree in person, number and gender.[4] Rather, person, number and gender are all subfeatures of a higher feature, labeled AGR (in GPSG) or INDEX (in HPSG). The information that a constituent has the agreement features third person singular feminine may be written as follows.

(3.17)
$$\left[\text{AGR}\left[\begin{array}{ll}\text{PER} & 3\\ \text{NUM} & \text{sg}\\ \text{GEN} & \text{F}\end{array}\right]\right]$$

---

[2] In GPSG, this is not stipulated in the PS rule itself, but falls out of a general principle, the Control Agreement Principle. In HPSG, this is a consequence of the Subcategorization Principle, which is discussed in the following section.

[3] I am assuming a familiarity with the concept of unification. For a basic introduction to unification. see [Charniak and McDermott, 1985]

[4] Gender agreement in English is less obvious than person and number, because verbs are not inflected for gender. But it is clear that gender information must co-vary in the subject and verb phrase, from reflexives that occur within the VP. This explains the ungrammaticality of *"He scratched herself."

BEST COPY AVAILABLE

English agreement may then be formulated in terms of a dependency involving this higher-level feature. The representation in (3.17) is referred to as an **Attribute-Value Matrix**, and is commonly used to represent complex feature structures.

3.1.3 Pollard and Sag's HPSG

GPSG uses complex feature structures to represent an arbitrarily large number of nonterminal categories. PS rules make direct mention of those aspects of the feature structure that are relevant for a given construction. There may be a large number of such PS rules; for example, in a sample grammar given in [Gasdar *et al.*, 1985a] for a fragment of English, there are approximately fifty rules for expanding a phrasal category as a lexical head and complements of the number and type which that head requires.

Like GPSG, HPSG uses complex feature structures to generalize nonterminals. In HPSG, rather than containing a bundle of unrelated feature-value pairs, each linguistic constituent consists of a single feature structure, encoding all of its syntactic properties. In addition, there is a generalisation over PS rules, so that a small number of generalized rules have the same generative power as an arbitrarily large number of fully specified PS rules. These generalized rules are called **schemata**. The exact inventory of schemata is still an open question, but [Pollard and Sag, In press] give seven which represent headed constructions.

One example of a schema is Schema 2, from Chapter Nine of [Pollard and Sag, In press].[5]

(3.18) `Schema 2 (revised version) a phrase with DTRS value of sort` *head-comps-struc* `in which the HEAD-DTR value is a lexical sign.`

This schema licenses any local subtree which has the following properties:

- The mother is phrasal.
- The mother has a feature called DTRS whose value is a certain type of object, namely *head-comps-struc*.
- The value of DTRS itself contains a feature HEAD-DTR. (The value of DTRS may contain other features as well, but these are not specified.)

---

[5] This chapter of [Pollard and Sag, In press] contains speculative ideas on modifications to the version of the theory found in the rest of the book, so this version of the schema is different from that given in Chapter One.

- The value of this HEAD-DTR feature is lexical.

There is much that this schema does not say. Most importantly, it does not say that the lexical element which is the value of the feature HEAD-DTR can subcategorise only for elements which are other daughters of the phrasal mother besides HEAD-DTR, or that these other daughters must be subcategorized for by the lexical element which is the value of the feature HEAD-DTR. These requirements are not stipulated in this schema because they are more elegantly captured in a general principle of the grammar, namely the Subcategorization Principle.[6]

(3.19) **Subcategorization Principle**

> In a headed phrase (i.e. a phrasal sign whose DTRS value is of sort *head-struc*), the SUBCAT value of the head daughter is the concatenation of the phrase's SUBCAT list with the list (in order of increasing obliqueness) of SYNSEM values of the complement daughters.

There is much in this principle which I will not explain here. What is relevant for the present discussion is that the principle allows schemata to be silent about how many complements are in a PS rule, and what feature structure information they contain. The lexical entry of the head specifies this information in the SUBCAT list, and the Subcategorization Principle ensures that in order for a structure to be licensed, it must have the appropriate complement daughters, as specified on the SUBCAT list of the lexical head.

The use of rule schemata rather than fully instantiated rules allows greater generalizations to be captured concisely in the grammar. As a result, HPSG grammars tend to be relatively small and simple. There are a small number of rule schemata; as previously mentioned, there are approximately seven for headed structures, plus several more for non-headed structures. There are on the order of a dozen general principles, such as the Subcategorization Principle given above, that further instantiate these schematic rules. In addition, there are lexical rules, which will be discussed below. These license lexical items of one type on the basis of a systematic relationship to lexical items of another type. For example, a lexical rule called "PASSIVE" licenses passive verbs on the basis of their active counterparts.

---

[6] The version of the principle given here is that found in Chapter One. This is reformulated in Chapter Nine as the Valence Principle, so that it applies to subjects and specifiers, as well as to complements.

As PS rule schemata become simpler, lexical items become correspondingly more complex. For example, in GPSG, a verb's subcategorisation list is implicit in its type. The verb "give" is listed in the lexicon as a V[3], and the following PS rule (from [Gazdar *et al.*, 1985a], p. 247) spells out the complements that such a verb takes.[7]

(3.20) VP → H[3], NP, PP[*to*]

In HPSG on the other hand, the entire subcategorisation list of "give" is spelled out in the feature structure of its lexical entry. The third person singular present form of this verb is given in Attribute-Value Matrix form (AVM) as follows (from [Pollard and Sag, In press], p. 13).

(3.21) gives

$$\left[\begin{array}{ll} \text{CAT} & \left[\begin{array}{ll}\text{HEAD} & \text{verb[fin]}\\ \text{SUBCAT} & \langle \text{NP[nom]}_{\boxed{1}[\text{3rd,sing}]}, \text{NP[acc]}_{\boxed{2}}, \text{NP[acc]}_{\boxed{3}} \rangle\end{array}\right] \\ \text{CONTENT} & \left[\begin{array}{ll} \text{REL} & \text{give}\\ \text{AGENT} & \boxed{1}\\ \text{GOAL} & \boxed{2} \\ \text{THEME} & \boxed{3}\end{array}\right]\end{array}\right].$$

Locating subcategorisation information in the lexicon rather than in grammar rules allows all of the fifty head-complement PS rules of GPSG to be captured by a single schema, Schema 2, plus the general principles of the grammar. Schemata do not have to specify the number or type of the complements, because the subcategorisation principle says that they have to correspond to what is in the SUBCAT feature of the head. Subcategorisation information is found in the lexicon.

The fact the a lexical item is so rich in information does not mean, however, that all of this information must be stipulated individually for each entry in the lexicon. Lexical items are organized in a hierarchy, so that information can be given in a more general and principled way. For example, there is a node on the lexical hierarchy denoting finite verbs. All entries which are dominated by this node are known not only to have the value "verb[fin]" for their CAT | HEAD feature,[8] but also to have a SUBCAT list whose first element is a noun phrase in the nominative case. This is a way of encoding the English-particular fact that finite verbs must have a nominative subject. The lexical hierarchy is further discussed in [Pollard and Sag, In press].

---

[7] In the rule, the category of the verb is given as 'H,' which is mnemonic for 'Head.' In PS rules that license headed structures, the category of the head is not specified, because it is predictable from a general principle, the Head Feature Convention.

There are many aspects of HPSG which I have not touched on here, such as the use of an obliqueness hierarchy to encode the complements of a verb, and some of the other general principles. But the description given here is sufficient for a basic understanding of the context within which the learning algorithm operates. Other aspects of HPSG will be mentioned below as they are relevant to the discussion.

3.2 The UNICORN Parsing System

In this section, I give a brief overview of the UNICORN system, which is based on Shieber's extension to Earley's algorithm. The discussion here focuses in particular on those aspects of the system that are relevant for the learning mechanism, which will be described in Chapters Six and Seven. For a fuller discussion of all aspects of Earley's algorithm and the extensions to it, see [Earley, 1970] [Shieber, 1985] and [Gerdemann, 1991].

3.2.1 Earley's Algorithm

Earley presents a general algorithm [Earley, 1970] which is able to parse sentences of any context-free grammar (CFG) in n^3 time. That is, the length of time required to parse sentences of a CFG increases at a rate which is linearly proportional to the cube of the rate at which the number of words in the sentence increases. The algorithm scans the input from left to right, and as each item is scanned, the algorithm constructs a **state set**, a set of the predictions made. Each state includes:

- the right hand side of a production (Phrase Structure rule)

---

[8] The notation "verb[fin]" is itself an abbreviation for the feature structure [[MAJ V] [FORM fin]], where "fin" stands for "finite."

- a point in that production, noted by a dot, representing the current position in the rhs of the production
- a backpointer to a position in the input string, where the category of the left-hand side (lhs) of the current production began
- a lookahead string for the successor of that production

The algorithm processes each state in a set with one of three procedures.

1. The **predictor** is invoked if there is a non-terminal to the right of the dot. It adds a new state for each alternative expansion of that non-terminal by a production rule, with the dot at the beginning of the rhs of the production. The lookahead is set to what follows the non-terminal on the rhs.
2. The **scanner** is called if there is a terminal to the right of the dot. It matches the terminal with the input, a process referred to as *lexical lookup*. If a match is found, the scanner adds the current state, S_i to the next state set, S_{i+1}, but with the dot moved over past the terminal. If S_{i+1} is null after S_i is all processed, then the string has no successful parse.
3. The **completer** is invoked if the dot is at the end of the production. It compares the lookahead string with the input. If it matches, the completer inspects the state set pointed to by the backpointer, retrieves all states with that string to the right of the dot, and adds those states, with the dot moved over, to the state set.

Earley's algorithm is similar to Knuth's LR(k)[9] except that [Knuth, 1965] uses a stack rather than a backpointer. If the grammar is unambiguous and reduced, then the completer takes n time, rather than n^2, so the complete algorithm takes n^2 time, rather than n^3. For bounded state grammars and LR(k) grammars, the algorithm works in linear time.[10]

Earley's original paper explains the algorithm in terms of a recogniser, but it can also be used to construct a parser. This is done by having the algorithm build up the tree as it goes, by keeping track of backpointers. This technique has been implemented in the UNICORN system.

---

[9] In this term, "L" stands for left-to-right parsing, "R" for a rightmost derivation, and "k" for a lookahead of variable length.

[10] Null productions make it hard to implement the completer.

3.2.2 Shieber's Extension to the Earley Algorithm

Earley's algorithm provides an efficient means of parsing sentences of context-free grammars. However, such grammars are not useful in accounting for the data of natural language. For example, as was discussed in the preceding section, it is not possible to formulate a CFG to account for long-distance dependencies, such as subject-verb agreement, without a prohibitively large number of nonterminals. Researchers in theoretical linguistics and natural language processing have therefore adopted formalisms that are more powerful than CFGs, such as the Unification-Based Grammars (UBGs) described above, in order to circumvent this weakness.

Shieber proposes a general technique for extending CFG parsing algorithms to the UBG formalisms [Shieber, 1985]. He demonstrates that the methods used in CFG parsing can be applied in UBG parsing, promising a degree of efficiency comparable to CF parsing algorithms.

Central to Shieber's proposal is the notion of *restriction*. Restriction provides a means for representing a potentially infinite set of nonterminals as a finite set of partially specified nonterminals, or generalised categories. A restrictor is a set of paths that the parser is instructed to 'pay attention' to.

For example, when parsing a noun phrase, having processed an initial determiner, the possible continuations include, among others, singular or plural noun.[11] If the determiner is "a," then the following noun must be singular. However, if the determiner is "the," then the number of the noun phrase is left open. If the restrictor contains a path for number information, then the scanner of Earley's algorithm will check the path for number information on lexical lookup. So following the determiner "a," it will retrieve only singular nouns. Without this path in the restrictor, all nouns will be retrieved for lookup. However, only those with singular number will successfully unify in the given production, due to the constraints imposed by the determiner. Therefore, the end result will be the same, regardless of the restrictor, but the process by which this result is achieved can differ greatly in efficiency.

It is up to the grammar writer to determine an appropriate restrictor. Restrictors can be specified giving various degrees of information, ranging from (trivially) no information at all, or just gross syntactic category (*e.g.*, noun, verb) to fine distinctions (*e.g.*, verb taking a nominative singular third person subject, one accusative plural object, and one dative object).

---

[11] I assume for the purpose of this example that singular and plural nouns are given separate lexical entries. This is not meant to be a theoretical claim, but merely a matter of efficiency.

The trade-off is that too little information in the restrictor leads to too many possibilities being explored, while too much information in the restrictor places a greater burden on the scanner, which must match lexical items being looked up with all of the specifications in the restrictor. I will speak of the information in the restrictor as delineating a syntactic category. The amount of information in the restrictor therefore determines how fine-grained a distinction is made among syntactic categories.

One weakness of Earley's original algorithm was the left recursion problem. Given a left recursive production, Earley's algorithm would predict the same category infinitely many times, leading to an infinite loop. Shieber's use of a restrictor in the prediction step eliminates this problem. The first category on the right-hand side of a left recursive rule is predicted only once, as a generalized category. It may then be *completed* arbitrarily many times, by repeated applications of the completion step to the same prediction. Shieber implements his extension to Earley's algorithm as a chart parser.

3.2.3 Optimizations in Unicorn

The learning system described here works in conjunction with Unicorn, a grammar-independent processing system which implements the principles of Unification Grammar. Unicorn was developed by Dale Gerdemann and Erhard Hinrichs [Gerdemann and Hinrichs, 1988] in the Computational Linguistics group of the Beckman Institute, at the University of Illinois.

Shieber confined his application of restriction to the first step, prediction, and then only to guarantee termination. Gerdemann and Hinrichs ([Gerdemann, 1991], [Gerdemann and Hinrichs, 1988]) recognized that restriction could be profitably applied to the remaining steps in an implementation of Shieber's extension to Earley's algorithm, and from this starting point implemented the Unicorn grammar processing system.[12]

3.3 The Advantages of Unification

While there have been a number of systems implemented to acquire new lexical items in a Natural Language Processing system, it is only recently that any have been specifically designed to take advantage of the unification-based approach. Such an approach has several advantages over previous language acquisition strategies, as described in [Lytinen and Hastings,] and [Gerdemann, 1991].

---

[12] "Unicorn" stands for 'Unification in the corn fields.'

Before the rise of unification-based grammar formalisms, perhaps the most widely used formalism for encoding grammars for Natural Language Processing was in Augmented Transition Networks (ATNs). In this section, I describe some of the advantages of unification-based grammar formalisms, as opposed to ATNs.

The primary advantage of unification-based formalisms over previous approaches to Natural Language Processing is modularity. The processing mechanism itself is separated from the grammar. This is in contrast to ATNs, in which the grammar and the processing mechanism are inseparably bound. This has several consequences. One is reversibility; unlike ATNs, unification-based NLP allows us to use the same grammar for both parsing and generation. Unification-based grammars are formulated with a declarative representation. Constraints can be satisfied in any order, making the process of parsing logically equivalent to that of generation. This allows us to test the understanding of the system by seeing whether the language it produces is correct.

Since UNICORN is a grammar- and lexicon-independent processing system, the learning mechanism is not bound to the form of the lexicon. In particular, it is able to operate with any kind of syntactic and semantic representation in the lexicon, so long as it is compatible with a unification-based formalism. This is why the use of existing linguistic knowledge is central to the acquisition of new lexical items. The learning mechanism has access to information in a lexical hierarchy, and creates new entries which reside at the proper node of the hierarchy. This allows the learning mechanism to be used for acquisition with lexicons of any format, so long as the hierarchical information is given.

Among the grammars currently being used with UNICORN are some which use a Montagovian semantic representation, with predicate-argument structure, and others with a Davidsonian semantics, with slot-filler notation. These representations, while both widely used in the literature, are very different, and provide good test cases for the learning mechanism. I argue in Chapter Four for the superiority of a Davidsonian semantic representation, but the learning mechanism is capable of interfacing with either, with only differences in relative efficiency. Similarly, grammars differ in their treatment of syntactic phenomena, and even their inventory of syntactic categories. Some grammars currently being used with UNICORN have a separate category for prepositions, while others treat prepositions as adjectives or adverbs that subcategorise for an NP. Again, the learner is not sensitive to these differences, so long as the appropriate hierarchical information is available.

In addition, the learning mechanism is flexible enough to work with lexicons of varying degrees of completeness. This is necessary in order to make it maximally useful for extension of both minimal 'core' lexicons, such as the Core Language Engine of [Carter, 1989], and lexicons that are designed to be relatively complete for a given domain. While many general applications may aim toward the use of an on-line dictionary, the use of a Natural Language Processing system in specific domains often requires a specialized, hand-coded lexicon. In both cases, if the system is to be used with unrestricted input, it will encounter lexical items not stored in the lexicon, and the acquisition mechanism must be able to learn them. This is possible because the system is not dependent on a particular representation of lexical entries.

Finally, unification gives us the ability to use partially specified lexical items, filling in more information about them as it becomes available. With such a system, it is possible to extract for a new lexical item only as much information as is available about it from a given context, and leave the rest unspecified, to be filled in on future encounters with the same lexical item. This allows the burden of acquisition to be shifted from the human user to the system.

3.4 Writing Grammars for UNICORN

The notation for grammars written with the UNICORN system closely follows that of [Gazdar and Mellish, 1989], with both rules and lexical items expressed in terms of path equations. In this section, I explain the notation of path equations, and describe the tools designed by Lake [Lake, 1991] for capturing hierarchical generalizations by means of macros, called definitions.

3.4.1 Path Equation Notation

Both production rules and lexical entries are represented in unification-based grammars as Directed Acyclic Graphs (DAGs). There are a number of ways of representing DAGs, which are appropriate for different purposes. One of these representations which is convenient for expressing feature values and structure sharing is path equations.

A path is a series of zero or more feature values, each of which is a top-level feature of the type that fills the value of the preceding feature. The top-level type in HPSG grammars is *sign*, which has the features `PHON`, `SYNSEM`, and `DTRS`. One example of a complete path in an HPSG grammar would be `<SYNSEM LOCAL CATEGORY HEAD FORM>`, also written as SYNSEM | LOCAL | CATEGORY | HEAD | FORM.

Path equations in Unicorn grammars serve one of two purposes. They either specify an atomic value which occurs at the end of a path, or specify that two paths share the same value, without stating what that value is. Structure sharing in DAGs is referred to as re-entrancy. The following entry for a common noun contains a number of atomic path equations, which have a path on the left hand side and an atomic value on the right. This lexical entry also contains several path equations denoting structure sharing. These have paths on both sides, and signify that the two paths share the same value, whatever that value may be.

```
(3.22) Word fender:
       <phon list first> = fender
       <phon list rest> = <phon eol>
       <synsem local category head maj> = n
       <synsem local category head maj> = <maj>.
       <synsem local category head form> = norm
       <synsem local category head case> = acc nom
       <synsem local category bar> = zero.
       <synsem local category marking> = unmarked
       <synsem local content index gend> = neuter
       <synsem local content index per> = third
       <synsem local content index num> = singular
       <synsem local content restr first relation> = fender
       <synsem local content restr first instance> =
              <synsem local content index>
       <synsem local content restr rest> = end
       <synsem local context list> = <synsem local context eol>
       <qstore list> = <qstore eol>.
```

Of the structure-sharing path equations above, the two involving the end-of-list feature (`eol`) are used to encode the mechanism of difference lists in the composition of certain features. This will not be further discussed here. A third re-entrancy unifies the value of the path SYNSEM | LOCAL | CATEGORY | HEAD | MAJ with a top-level feature, also called MAJ. The former is the path which encodes major category according to HPSG theory, while the latter is required by the implementation of Unicorn itself.

The remaining re-entrancy is used to represent facts of HPSG-style semantics. The feature structure encodes the fact that the semantic content of a common noun such as "fender" consists of an index and a set of restrictions. Unicorn grammars cannot currently encode unordered sets, so we use the list notation, specifying an elements position in a list via the FIRST and REST features.[13] The first element in the restriction set is a Parameterised State of Affairs (PSOA), consisting of a relation and an instance. This relation is lexically specified, somewhat trivially, as that of being a fender,[14] and the INSTANCE is structure-shared with the INDEX. Intuitively, the word "fender" is used to denote something which carries third person singular neuter agreement, and which is an instance of the semantic concept of 'fender.'

3.4.2 Definitions

As mentioned above, HPSG production rules are highly schematised. The information that might be found in such rules in other theories is here encoded with either general principles, or in lexical items themselves. Lexical items participate in a hierarchy, which makes it possible to specify information at more general levels, rather than stipulate it for each lexical item.

Both means of encoding information in a more general way are accomplished with Unicorn grammars by means of definitions. This is a capability for encoding macros, added as one of a number of user interface enhancements to Unicorn by Lake [Lake, 1991]. The mechanism of definitions is conceptually very simple, yet it allows significant generalisations to be captured concisely. I will here give one example of the use of definitions for each type of information.

---

[13] This notation for lists was originated by Gazdar *et al.* in [Gazdar *et al.*, 1985b].

[14] The notation for atomic values here hides some very important type distinctions. The value of the path PHON | LIST | FIRST is actually the phonological or orthographic representation of the word. Similarly, the value of the semantic RELATION feature is actually the semantic concept of being a fender, which might be written in Montague semantics as ˆfender′.

3.4.2.1 Lexical Hierarchical Information

Lexical information can be specified for a word at any of a number of points in the lexical hierarchy. There is information that all words share, such as having their phonology represented by a difference list, whose REST is equal to the EOL feature, and requiring a top-level feature called MAJ which is structure-shared with the major category feature found within the HEAD feature structure. All substantive words, including the major categories of noun, verb, adjective and preposition, have a bar-level of zero, while function words do not participate in the bar-level hierarchy. This information is represented below.

```
(3.23) define word:
       <phon list rest> = <phon eol>
       <synsem local category head maj> = <maj>.
```

```
(3.24) define function:
       word
       <synsem local category bar> = nil.
```

```
(3.25) define subst:
       word
       <synsem local category bar> = zero.
```

A verb is a substantive word whose major category is "v." In addition, a verb in English has exactly one subject, about which a great deal of information is known. This subject must itself be saturated; that is, the subject itself is not missing any complements, specifiers, or subjects, and is in the nominative case. In general, the subject has a major category of "n,", and has a form of "normal."[15]

```
(3.26) define verb:
       subst
```

---

[15] These latter two statements may have exceptions, as in the following sentences. "Under the sofa would be a good place to hide the dirty dishes." "There is a unicorn in the garden." Verbs which allow such subjects will be specified individually in the lexicon, rather than inheriting their path equations from the general definition verb.

```
<synsem local category head maj> = v
<synsem local category subj first synsem local category head
        maj> = n
<synsem local category subj first synsem local category head
        case> = nom
<synsem local category subj first synsem local category head
        form> = norm
<synsem local category subj first synsem local category
        comps> = end
<synsem local category subj first synsem local category subj>
        = end
<synsem local category subj first synsem local category spr>
        = end
<synsem local category subj first synsem local category bar>
        = two
<synsem local category subj first synsem local conj> = nil
<synsem local category subj rest> = end
<synsem local context list> = <synsem local context eol>
<qstore list> = <qstore eol>.
```

There is also a node in the hierarchy cross-cutting the division according to syntactic categories, which captures generalizations about transitivity. The notion of a lexical item that is transitive, taking a single nominal complement, is expressed as follows. This includes not only transitive verbs, but prepositions and certain adjectives as well.

(3.27)
```
define transitive-n:
<synsem local category comps first synsem local category
        head maj> = n
<synsem local category comps first synsem local category
        head form> = norm
<synsem local category comps first synsem local category
        head case> = acc
<synsem local category comps first synsem local category
        comps> = end
<synsem local category comps first synsem local category bar>
```

```
          = two
    <synsem local category comps rest> = end.
```

The notion of a transitive verb taking a single nominal complement is then simply expressed as the combination of these two definitions, with the addition of a path which is unique to this category. This path specifies how its semantic value combines with that of the categories it combines with, which is different from the method of combination for other transitive categories.

```
(3.28) define verb-comp-n:
       verb
       transitive-n
       <synsem local content restr> = end.
```

Particular lexical transitive verbs are instances of this category. The information that must be specified in addition to this definition is exactly what properly belongs in the lexicon. This includes the phonological form of the lexical item and the parameterised state of affairs which constitutes a verb's semantic content. The following is the representation for the base form of this verb.[16]

```
(3.29) Word hit:
       verb-comp-n
       <phon list first> = hit
       <synsem local category head form> = base
       <synsem local content relation> = hit
       <synsem local content hittee> =
             <synsem local category comps first synsem local content>
       <synsem local content hitter> =
             <synsem local category subj first synsem local content>.
```

---

[16] Intuitively, the base form of a verb is the infinitive without the word "to." The base form is that which does not bear any tense or agreement information.

The information specified here is exactly what belongs in the lexical entry of a particular word. The first line gives its class membership, `verb-comp-n`, from which a great deal of other information is inherited. The only things stipulated lexically are the phonological and morphological form, the semantic relation denoted by the verb, and the roles in that relation played by the arguments of the verb. Thus the mechanism of definitions enables the lexicon writer to specify each kind of lexical information at the appropriate level of generality.

3.4.2.2 Rule Schemata

One important principle of HPSG grammar is the Head Feature Principle (HFP).[17] A particular subset of the features of syntactic categories are designated as HEAD features. These include syntactic category information for all categories, case for nouns, the category of the modified noun for adjectives, etc. There is a node of feature structure labeled HEAD, which dominates all of these features. The HFP is given as follows (from [Pollard and Sag, In press], p. 18).

(3.30) **Head Feature Principle**

> The HEAD value of any headed phrase is structure-shared with
> the HEAD value of the head daughter.

This is encoded as a definition in a UNICORN grammar, using a path equation that specifies the full paths that share structure. In path equations that describe a production rule, the term '`x0`' denotes the element on the lhs of the rule. The elements on the rhs are denoted by '`x1`,' '`x2`,' etc., from left to right.

(3.31)
```
define hfp:
<x0 synsem local category head> =
      <x0 dtrs headdtr synsem local category head>.
```

---

[17] This performs roughly the same work as the Head Feature Convention (HFC) of GPSG, though the inventory of Head Features differs for the two theories.

The HFP applies to any headed structure, which includes all of the schemata given in [Pollard and Sag, In press]. The formulation of the schema for any headed structure therefore includes the definition of the HFP. All of the path equations specified in this definition (in this case, just one) are then included in the compiled version of the grammar.

Schemata themselves are given in the grammar as definitions. The following schema, which licenses all rules for expanding a phrasal category as a lexical head and a single complement, shows the HFP, as well as other general principles and specific path equations.

(3.32)
```
define comp-head-one:
hfp
context2
qip2
phonHeadComp1
<x0 synsem local category subj> =
      <x0 dtrs headdtr synsem local category subj>
<x0 synsem local category comps> = end
<x0 synsem local category bar> = one
<x0 synsem local conj> = nil
<x0 synsem local content> =
      <x0 dtrs headdtr synsem local content>
<x0 dtrs compdtrs rest> = end
<x0 dtrs headdtr synsem local conj> = nil
<x0 dtrs headdtr synsem local category comps first synsem> =
      <x0 dtrs compdtrs first synsem>
<x0 dtrs headdtr synsem local category comps rest> = end
<x0 dtrs headdtr synsem local category bar> = zero.
```

UNICORN cannot, however, work directly with schemata. It must have instantiated rules to use for parsing. These rules are instances of schemata, and stand in the same relation to schemata that schemata do to definitions. In general, the only information that an instantiated rule must specify in addition to that of the schemata is the syntactic category of some of the

BEST COPY AVAILABLE

elements,[18] the relation of each of the daughters to the head. The following rule is an instance of the above schema.

(3.33)
```
v → v x2:
comp-head-one
<x0 dtrs headdtr> = <x1>
<x0 dtrs compdtrs first> = <x2>.
```

The `comp-head-one` definition in this rule causes it to inherit all of the path equations from this definition, including the definitions that it contains, such as `hfp`.

Here again, we see that the rule itself specifies only that information which is unique to this rule. All else is inherited directly from a schema, and thus indirectly from general principles of the language.

---

[18] Strictly speaking, it is not necessary to instantiate any of the syntactic categories. However, for reasons that I will not discuss here, UNICORN operates more efficiently if the category on the lhs of the rule is instantiated, as well as the initial lexical head on the rhs.

Chapter 4

Initial Knowledge Used as the Basis for Learning

No learning system can begin with a totally blank slate. As the literature review in Chapter Two demonstrates, even those systems which claim to build a lexicon from scratch start with a great deal of knowledge about the structure of possible lexical items, as well as the grammar of the language. The approach to lexical acquisition that I am taking presupposes not only a grammar and a hierarchy of lexical types, but also a knowledge base that is assumed to be complete for the restricted domain in which learning occurs. These assumptions are made in order to draw as clean a separation as possible between linguistic and non-linguistic learning, as well as to delimit the scope of the task addressed in this thesis.

In this chapter, I describe those components of the system which were encoded in order to provide an environment within which learning can occur. This includes the grammar, the real-world knowledge base, and the interface between the two. Along the way, several important issues of representation arise, and I discuss the motivation for particular choices that I have made in the implementation.

The first section describes the grammar used with the learning mechanism, focusing on both its similarity to and divergence from the version of HPSG set forth in [Pollard and Sag, In press]. The second section describes the real-world knowledge base used with the system, how it is interleaved with the parsing algorithm, and its use in disambiguation.

4.1 The Grammar Used in Learning

Unicorn is a grammar-independent processing system, as described in Section 3.2. Since the initial development of the system, a number of unification-based grammars have been constructed to be used with Unicorn. One such grammar is currently being used for generation of English sentences in a driving instruction system, as described in [Frisch *et al.*, 1990]. Other grammars are presently being developed, both for applications with particular NL-related tasks, and to carry out linguistic research in English and other languages.

In this section, I describe the grammar used with the learning mechanism. Although this grammar was developed with the task of lexical acquisition in mind, it was also designed to capture a wide range of syntactic phenomena of English. The coverage is sufficient to handle a body of naturally occurring text, consisting of fifteen police reports of traffic accidents, each report containing approximately fifteen sentences. This corpus and the parsing results achieved for it are given in Appendix C.

The grammar described here is modeled closely after the version of HPSG set forth in [Pollard and Sag, In press]. I will not summarise this entire work, but only comment on how certain aspects of this theory have been implemented in the grammar. I also point out particular ways in which my implementation differs from what might be expected from the theory.

4.1.1 The Implementation of HPSG2 in Unicorn

This section describes various aspects of the implementation of HPSG as outlined in [Pollard and Sag, In press]. While the implementation is intended to faithfully adhere to the principles described there, certain constraints imposed by Unicorn itself have shaped the particular method of encoding the principles in ways which are explained here.

4.1.1.1 Application of General Principles

In [Pollard and Sag, In press], there is no notion of Phrase Structure rules. A local subtree is licensed if and only if it conforms to all of the applicable general principles of the language, and fits the template of one of the rule schemata.

However, Unicorn requires Phrase Structure rules for parsing or generation. Therefore, a Unicorn grammar contains PS rules, constructed as described in the previous chapter, using the definition facility. A PS rule in a UNICORN grammar consists of an inheritance link to the schema which it instantiates, similar links to all the general principles which apply to it, and any additional information idiosyncratic to the rule, not found in the schema or any general principle from which the rule inherits.[1]

A reasonable question at this point might be, if the general principles are truly general, then why not simply require that every PS rule conform to all of them, allowing those that are applicable to apply their constraints, and the others to apply vacuously? The answer is that the way the principles are formulated contains an implied *if-then*. As UNICORN rules themselves cannot handle *if-then* constructions, it is up to the grammar writer to include the principles only when they are applicable. This may be seen in the formulation of the Head Feature Principle (from [Pollard and Sag, In press], p. 24), repeated here.

(4.1) `Head Feature Principle`

> `The HEAD value of any headed phrase is structure-shared with`
> `the HEAD value of the head daughter.`

The implied *if-then* here is in the phrase "any headed phrase." This is equivalent to saying "If something is a head phrase, then its HEAD value is..." If something is not a headed phrase, then it does not have a HEAD value. But the UNICORN rules cannot tell this. The effect of the HFP is to require that any rule which adheres to it has a HEAD value structure-shared with the HEAD value of the head daughter. If this principle were applied to non-headed phrases, it would specify paths for them which they should not have. Therefore, this principle must be stipulated to apply to all and only those rules which license headed structures. Similar observations apply to the other general principles of the grammar, whether universal or language-specific.

4.1.1.2 Word Order and the PHON Feature

As mentioned in Chapter Three, the PHON feature, mnemonic for 'phonology,' is designed to encode the surface form of a linguistic entity. It is envisioned that in future work, this will

---

[1] It is possible to write PS rules without making use of the definition facility. In this case, all of the path equations that a rule adheres to must be stipulated in the formulation of the rule itself, rather than being inherited from a schema or from general principles.

66

actually be used to describe phonological processes in a meaningful way, but this has not been the focus of [Pollard and Sag, 1987] and [Pollard and Sag, In press]. Here, this feature is simply used to store the orthographic form of words and phrases, and the value of the mother of a phrase is the concatenation of the values of its daughters.

However, in Chapter Seven of [Pollard and Sag, 1987], principles of constituent ordering are described as interacting with the PHON feature. I here describe my implementation of this feature, and show how it can be used to encode these ordering principles. The encoding of this feature also provides an example of the use of **difference lists**, a data structure which is also used to implement other aspects of linguistic representation.

The value of the PHON feature is a list, whose elements are the words of the string. Ideally, the PHON value of the mother, which is the concatenation of the PHON values of the daughters, should be determined by an append operation. However, UNICORN grammars cannot make use of the append operation, and so I have encoded the PHON feature (and several other features whose values are determined by concatenation) by means of *difference lists*.[2]

In such a representation, every linguistic entity has two paths in its PHON feature: LIST, which holds the list of words making up the phonological value of that entity, and EOL, a dummy value representing the end of the list. The list which is the value of the LIST feature itself consists of two paths: FIRST, which may take an atomic value or a list value, and represents the first element of the list; and REST, which necessarily takes a list value, possibly the empty list, and represents the remainder of the list.[3] Lexical items have a LIST feature whose FIRST value holds the phonological form of that item, and whose REST value is structure-shared with the EOL. The phonological forms of the lexical items "cross" and "Elm" are given in (4.2) and (4.3), respectively.[4]

```
(4.2) Word cross:
     <phon list first> = cross
     <phon list rest> = <phon eol>
```

---

[2] I am grateful to Dale Gerdemann for suggesting this method of encoding the PHON feature.

[3] This method of encoding lists in AVM feature structures is due to [Gazdar *et al.*, 1985b].

[4] The single quotes on 'Elm' are necessary when the orthographic form of a word or phrase includes capital letters or non-alphabetic characters, such as dashes, or the spaces in multi-word lexical items. In general, I write feature names in capital letters in the text, in accordance with Pollard and Sag, but in lowercase in path equations, as this is the form required for processing in UNICORN.

BEST COPY AVAILABLE

(4.3)
```
Word elm:
    <phon list first> = 'Elm'
    <phon list rest> = <phon eol>
```

Rather than explain the theory of how difference lists work, it will be clearer to give an example. The simplest method of specifying the PHON features of the mother in terms of those of the two daughters is illustrated in (4.4). This is, in fact, the rule used in the grammar for determining the PHON features of the mother in the default case. As described in the previous chapter, 'x0' denotes the entity on the lhs of the PS rule, 'x1' the first entity on the rhs, and 'x2' the second entity on the rhs.

(4.4)
```
define phon2:
    <x0 phon list> = <x1 phon list>
    <x1 phon eol> = <x2 phon list>
    <x0 phon eol> = <x2 phon eol>
```

If "cross" and "Elm" are the daughters in a local subtree which is licensed by a rule which includes a link to the definition phon2, then the mother has as the value of its PHON feature a list representing the string "cross Elm," as encoded below. This follows straightforwardly from the path equations in (4.2), (4.3) and (4.4).

(4.5)
```
<phon list first> = cross
<phon list rest first> = 'Elm'
<phon list rest rest> = <phon eol>
```

The reason that EOL is called a dummy value is that it is never instantiated to a particular atomic value. It is always structure-shared with the value at the end of the LIST of the PHON feature. Thus EOL is simply a bookkeeping feature, whose value need never be instantiated.[5]

---

[5] There is a requirement in HPSG that all top-level signs, *e.g.*sentences, must have all of their values instantiated. This requirement can easily be met by stipulating that a well-formed sentences has a path PHON | EOL with some particular atomic value.

The result is that the PHON value of the mother is a flat list representing the concatenation of the list values of its daughters.

Although the definition for the concatenation of the PHON values of the daughters given in (4.4) is the simplest, it is not the most general from a linguistic point of view. This is because it makes reference to the order in which the daughters happen to be specified in a PS rule. However, as pointed out in [Gazdar *et al.*, 1985a], there are more general properties of constituent ordering in language, which can be captured by separating out the Immediate Dominance (ID) properties of PS rules from the Linear Precedence (LP) principles. On this view, the fact that the daughters of a production rule are specified in a particular order should not be construed as claiming that that reflects their surface order in structures licensed by that rule.

One example of an LP rule is that, in English,[6] lexical heads precede their sister complements. For example, a lexical verb in English precedes all of its sister complements. This LP rule is encoded in the following definition.

(4.6)
```
%% In a head-comp structure, the lexical head goes first.
%% This version is for heads with exactly one comp dtr.
define phonHeadComp1:
<x0 phon list> = <x0 dtrs headdtr phon list>
<x0 dtrs headdtr phon eol> = <x0 dtrs compdtrs first phon list>
<x0 phon eol> = <x0 dtrs compdtrs first phon eol>.
```

If we assume that the rule licensing the combination of "cross" and "Elm" to form "cross Elm" is an instance of the head-complement schema, then the definition in (4.6) produces exactly the same feature structure for PHON as (4.4) above. On the surface, (4.6) appears to be more complex; it contains the same number of path equations, but many of the corresponding paths are longer in (4.6) than in (4.4). However, this formulation captures the fact that it is not just an arbitrary stipulation of a particular rule or schema that put the lexical head daughter before the complement. Rather, this is a general property of English.

---

[6] Like all LP rules, this is language-particular.

Ideally, it should be possible to encode all of the constituent ordering facts about a language with such rules. Where constituents can occur in either order, both orders are specified as possibilities. The following LP rules of English have been encoded in the grammar.

1. In a head-subject structure, the head goes last.
2. In a head-complement structure, the lexical head goes first.
3. In a head-specifier structure, a non-V head goes last.
4. In a head-specifier structure, a V head goes first.
5. In a head-marker structure, the head goes last.
6. In a head-adjunct structure with a non-V head and a lexical adjunct, the adjunct goes first.
7. In a head-adjunct structure with a non-V head and a phrasal adjunct, the adjunct goes last.

It is clear that some of these LP rules are more concise than others. For head-subject, head-complement and head-marker structures, it is necessary to make reference only to the position of the head with respect to the other constituents. With head-specifier structures, the category of the head is also relevant, with verbal heads behaving differently from others. Head-adjunct structures are the most complicated, with both the category of the head and the lexicality of the adjunct having to be taken into consideration.

In fact, the data concerning adjuncts are not entirely clear. The rules for constituent ordering of head-adjunct structures given above are intended to explain the difference between orderings such as those in (4.7) and (4.8).

(4.7) a. John spied a polished coin.

b. * John spied a coin polished.

(4.8) a. John spied a coin polished to a high sheen.

b. * John spied a polished to a high sheen coin.

However, it may be that "lexical" is not the correct term to capture the class of adjuncts that can precede nouns. Examples such as (4.9) indicate that the relevant measure may be something more like complexity or heaviness.

(4.9) a. John spied a very thoroughly polished coin.

b. ? John spied a coin very thoroughly polished.

It is not my purpose here to solve all of the open issues of constituent ordering in English, but merely to indicate how any solution could be encoded into the representation scheme outlined here. Those rule orderings which can clearly be captured by reference to the head and its syntactic category have been encoded into the grammar by means of definitions similar to (4.6) above. For others, the grammar currently uses the default rule, (4.4), until more general rules can be determined.

4.1.2 Differences from HPSG2

In this section I outline some choices made in implementing the grammar which either differ from the exposition of HPSG given in Pollard and Sag, or, where Pollard and Sag offer alternatives, have been made in favor of one alternative over the others.

4.1.2.1 Partitioning the SUBCAT List

Throughout the development of HPSG, there have been two positions on the structure of the subcategorisation list, with one or the other being advocated by different authors at different times. The first, which is predominant through most of [Pollard and Sag, 1987] and [Pollard and Sag, In press], is that there is a single subcategorisation list, called SUBCAT, which contains an ordered list of all the elements that the given category selects for. The ordering is according to obliqueness, with the least oblique subcategorized element either coming first or last on the list.

The second position is that instead of (or possibly in addition to) a single SUBCAT list, there are three separate valence lists. Two of these, SUBJS and SPECS, contain the subject and specifier that the element selects, and are restricted (in English, at least) to lists of at most one element each. The third, COMPS, contains the complements that the element selects, and may have any number of elements (though there are rarely more than two).

This second position is mentioned in [Pollard, 1989], and is later tentatively adopted in Chapter Nine of [Pollard and Sag, In press]. Very briefly, one argument in favor of this position is as follows. If there is just one valence list, then subjects and complements are the same kind of thing. A subject is not a special type of argument, but merely defined as the least oblique element on the SUBCAT list. However, this entails that if an element has only one element on its SUBCAT list, this element is a subject. This in turn means that what are generally referred to as the objects of prepositions are actually their subjects. However, they do not receive the case marking that would be expected of subjects, but of objects.

By maintaining separate valence lists for subjects and complements, we have a means of treating subjects differently from objects, while still allowing for categories, such as prepositions, which have complements but no subjects. It is this view of three separate valence lists that I adopt here.[7]

4.1.2.2 Order of Obliqueness

Whether there is one valence list or three, the elements on each list occur in order of obliqueness. It is merely a matter of exposition whether this order is of increasing or decreasing obliqueness. In [Pollard and Sag, 1987], the ordering is most oblique first. This was reported to be confusing, and so was changed to least oblique first in [Pollard and Sag, In press].

In the grammar, I have chosen the ordering of most oblique first, for reasons of efficiency rather than expository clarity. An item with more than one element on a valence list necessarily combines with the most oblique first. When using an AVM representation of feature structures representing lists by means of FIRST and REST, it is far easier to refer to the first element of a list than to the last element. Therefore, by keeping the most oblique element at the beginning of the list, it is much simpler to specify that the head combines with this element.

4.1.2.3 Semantic Content of Verbal Arguments

In [Pollard and Sag, In press], the semantics of the arguments of verbs is given by the path `<synsem local content index>`. However, I have instead given their semantics by the path

---

[7] Chapter Nine of [Pollard and Sag, In press] leaves open the possibility that there is also a single valence list, SUBCAT, whose value is the concatenation of the values of the three separate lists. While not explicitly rejecting this option, I have not found it to be necessary in formulating the grammar given here.

`<synsem local content>`, so that the restrictions on an NP's semantics are reflected in the semantics of the verb.

In HPSG, the semantic value of a noun, and therefore of an NP, is given as an index and a set of restrictions. The semantics of the verb, and therefore of the sentence of which the verb is the head, is a PSOA, with a relation and the appropriate number of arguments to that relation. In [Pollard and Sag, In press], the semantic type of the arguments is `index`, structure-shared with the index of the appropriate item on the SUBCAT list. However, this means that the set of restrictions on the NP does not occur anywhere in the semantic structure of the verb, and therefore of the sentence. For example, in sentence (4.10), the semantics of the sentence would be as in (4.11).

(4.10) A green book fell.

(4.11)
$$\begin{bmatrix} \text{CONTENT} & \begin{bmatrix} \text{PSOA} & \begin{bmatrix} \text{RELATION} & \text{FALL} \\ \text{FALLER} & \begin{bmatrix} \text{PERSON} & \text{THIRD} \\ \text{NUMBER} & \text{SINGULAR} \\ \text{GENDER} & \text{NEUTER} \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

This semantic representation contains only a PSOA with the FALL relation and a third singular FALLER. There is no information about GREEN or BOOK on the top-level node at all.

This difference between my representation and that of [Pollard and Sag, In press] on this point arises from a difference in goals. They are interested only in a truth-conditional theory of semantics, while I am also interested in building up an understanding of the meaning of the sentence. The restrictions on the NP contributes to the identification of its referent, but not to truth-conditions. Therefore, these restrictions are not necessary for strict truth-conditional semantics, but are necessary for the kind of understanding I wish to achieve.

There is an additional possible complication in the way I haven chosen to encode the semantics of verbal arguments. The features INDEX and RESTR together define the category `npro`, for 'non-pronominal.' However, other kinds of nominals can also be arguments of verbs, including pronouns and proper nouns. Each of these has its own type of semantics, which must then be allowed for as an option for the semantics of verbs.

An alternative would be to add a set of restrictions to the semantics of the verb, inside CONTENT but external to the PSOA, and specify that the restriction set of a verb is structure-shared with that of its arguments. I explore this alternative further in Section 3 of this chapter.

4.1.2.4 Bar-Level Representation

The grammar I have implemented includes an explicit encoding of bar-levels, and does not use the LEX feature. This is in accordance with some work in HPSG, such as [Pollard, 1989], and with the notation used in much of [Pollard and Sag, In press], though it differs from what is there claimed to be the actual feature structure of the theory.

In Chapter One of [Pollard and Sag, In press], (p. 23), it is claimed that whatever work had been done by appealing to bar-levels could be done just as well with a combination of the LEX feature and the SUBCAT list. Notation such as N′ is not meant to encode a literal representation of a bar-level of one, but rather an abbreviation for a nominal category with the feature-value [LEX minus] and a non-empty SUBCAT list (or perhaps "not-necessarily-empty," since plural nouns may have an empty SUBCAT list, but it may still be necessary to be able to refer to them at two levels).

However, the elimination of explicit bar levels in favor of the LEX feature and valence lists comes only at the expense of a number of empty categories. This is a direct result of the distinction between **non-empty** valence lists and **not-necessarily-empty** valence lists. This entails a corresponding distinction between **empty-because-filled** valence lists and **lexically-empty** valence lists, and there are cases where grammar rules must be able to refer to this difference. For example, the grammar should license (4.12) but not (4.13).

(4.12) the red wagons

(4.13) * red the wagons

At first, this does not seem to present a problem. The grammar must simply say that prenominal adjectives, such as "red," modify nominal categories whose SPR list is non-empty. However, this is not sufficient, because plural nouns may lexically have an empty SPR list; that is, the grammar must also license sentences such as (4.14).[8]

(4.14) Red wagons are fun.

Therefore, the "red" in (4.14) cannot select for a noun with a non-empty SPR list; the SPR list of "wagons" in this example is empty, just as that of "the wagons" in 4.13 is. Prenominal adjectives have to modify Ns that either have a non-empty SPR list, or a lexically-empty SPR list, but not an empty-because-filled SPR list. However, an empty SPR list itself does not specify how it came to be empty, which is the problem.

In fact, "wagons" (the lexical entry with an empty SPR list) does not differ from "the wagons" by any valence feature. Therefore, an explicit representation of bar-level is necessary in order to distinguish this entry for "wagons" from "the wagons," and thus to allow (4.14) while disallowing (4.13). Specifying that prenominal adjectives modify an N with bar-level one makes the correct predictions.[9]

This issue is brought up only briefly in [Pollard and Sag, In press] (p. 44), and is there left as open question. However, it was brought up for discussion in a workshop on HPSG at Stanford (July, 1992). There, Sag (p.c.) rejected the notion of explicit bar-levels in HPSG feature structure. In their place, he argued that all lexical nouns, including plurals and mass nouns, necessarily have a single element in their SPR list. For plurals and other nouns where there is no specifier on the surface, there is a null specifier, which combines with nominals in the same way as any other specifier, but has no phonological form. The semantics of the null specifier would then be such as is needed to give these forms their desired meaning.

The decision to encode bar-level explicitly in the grammar is, then, motivated at least partly by considerations of efficiency. It may well be that an analysis invoking null specifiers can be made to work for an analysis of English. However, the use of any phonologically null category in unification-based grammars tends to make parsing much slower. This is because something that is phonologically null vacuously satisfies any next input string. Therefore, a null lexical item is tried wherever an item of that category is predicted, including a large number of places where it is not a reasonable guess.

---

[8] In [Pollard and Sag, In press], prenominal adjectives are specified as modifying an element of category N′, but it is not clear what this abbreviation stands for, or how it applies to plural nouns.

[9] This seems to be hinted at for German on p. 29 of Chapter Nine, but it is not clear to me whether in this discussion bar-levels are regarded as actual entities, or merely as convenient abbreviations.

A phonologically null element of a single category, such as a specifier that selects for nominals, might not by itself spell disaster for a unification-based parser. But taking the position that such elements should be invoked for all cases in which the LEX feature and the valence lists do not provide sufficient specification would lead to a large number of such items. In particular, there must be one phonologically null item for every selected category for which the following situation obtains: (i) phonologically realised elements of that category may be selected for by some elements of another category, but (ii) not all elements of the selecting category must select a (phonologically realised) element of the selected category, and (iii) whether or not the selecting category has found all of this type of selected element makes a difference as to whether the selecting element may itself be selected at a higher level of the tree. In addition to phonologically null determiners, this forces the grammar to include null specifiers of other categories, and possibly null complements as well.

It was recognized at this workshop that having a large number of such null categories is highly undesirable, and they were christened "creatures of darkness" by Sag. However, an explicit encoding of bar-levels was also seen as undesirable, because then, bar-level and valence lists, while clearly related, would both have to be primitives of the grammatical representation. In the face of two undesirable alternatives, I am choosing to represent bar levels explicitly. Although this duplicates much of the work of the valence lists, it is by far the more efficient choice for implementing a parser of an HPSG grammar.

In the grammatical representation which I have adopted, then, bar level is not strictly predictable from the valence lists, but there is still a close correlation between them. A bar-level of zero denotes a lexical item, and thus replaces the LEX feature of [Pollard and Sag, In press]. A category of bar-level one is one which has found all of its complements, while a category of bar-level two has found everything it selects for via SPR. The SUBJ list makes no difference in bar-level.[10]

The difference between these two conceptions of the relation between bar-level and valence lists can be shown by a simple example. In the sentence "Fish swim," the lexical entry for the word "fish" has all three valence lists empty. The tree structure for this sentence with

---

[10] In Jackendoff's original formulation of X-bar syntax [Jackendoff, 1977], there is a bar-level three, which, translating into the current view of valence lists, would denote a category whose SUBJS list is empty. However, unlike the other bar-levels, I have not found any place where the grammar needs to make reference to a bar-level of three.

```
             S
            / \
           /   \
          N     V
 [SUBJS <>]   [SUBJS <N>]
 [SPRS  <>]   [SPRS  <> ]
 [COMPS <>]   [COMPS <> ]
   / \             /\
  /   \           /  \
•  [SUBJS <•>]   •  [SUBJS <N>]
   [SPRS  <> ]      [SPRS  <•>]
   [COMPS <> ]      [COMPS <> ]
     / \                   /\
    /   \                 /  \
   •  [SUBJS <•>]  [SUBJS <N>]  •
      [SPRS  <•>]  [SPRS  <•>]
      [COMPS <> ]  [COMPS <•>]
          / \         swim
         /   \
[SUBJS <•>]   •
[SPRS  <•>]
[COMPS <•>]
  fish
```

Figure 4.1: Representation of "Fish swim" without bar-levels.

phonologically null elements is shown in Figure 4.1, while Figure 4.2 shows the structure for the same sentence with an explicit bar-level representation.

Assuming phonologically null elements, since both N and V are categories that can take SUBJS, SPRS and COMPS lists, they must always do so. This leads to a large number of empty categories in the grammar.

4.1.2.5 Limitations of Coverage

There are several aspects of feature structure discussed in [Pollard and Sag, In press] which I have either not implemented at all, or only in a rudimentary way. These include the context, quantification, and filler-gap constructions.

Context includes information about the speaker and addressee of an utterance, background assumptions, and other pragmatic factors. I have encoded some of this straightforwardly in

```
              S
             / \
            /   \
           N     V
[SUBJS <>]  [SUBJS <N>]
[SPRS  <>]  [SPRS  <> ]
[COMPS <>]  [COMPS <> ]
[BAR  TWO]  [BAR  TWO ]
    |           |
    |           |
[SUBJS <>]  [SUBJS <N>]
[SPRS  <>]  [SPRS  <> ]
[COMPS <>]  [COMPS <> ]
[BAR  ONE]  [BAR  ONE ]
    |           |
    |           |
[SUBJS <>]  [SUBJS <N>]
[SPRS  <>]  [SPRS  <> ]
[COMPS <>]  [COMPS <> ]
[BAR ZERO]  [BAR ZERO ]
  fish        swim
```

Figure 4.2: Representation of "Fish swim" with bar-levels.

---

the grammar. However, I have formulated the grammar rules such that the CONTEXT of the mother of every local subtree is the union of the CONTEXT features of the daughters. This is known to be too strong, as is argued in Chapter Eight of [Pollard and Sag, In press]. Pollard and Sag mention several alternative approaches to this issue, but a complete solution is not given. As this is not the focus of the present work, I do not deal with it further.

Similarly, the treatment of quantification in the grammar is rudimentary at best. Pollard and Sag sketch the beginnings of a treatment of quantification in English, but there are a number of complicating factors. In addition, an implementation of as much of an analysis as they do give would require mechanisms for handling feature structures that do not currently exist in UNICORN, such as the ability to construct a list which is the union of two other lists minus the elements of a third list. Rather than attempting to solve all of the issues involved here, I have given determiners enough information to produce correct syntactic parses, but little more.

Because of the difficulties with phonologically null elements discussed in the previous subsection, I have deliberately not focused on filler-gap constructions. Therefore, the treatment of relative clauses is far short of what is outlined in [Pollard and Sag, In press]. However, the grammar is able to parse sentences containing subject relatives, which are the only kind appearing in the corpus of text being investigated.

4.1.3 Additions to HPSG2

In this section, I briefly outline the implementation of some aspects of English grammar that are not described in detail in the works of Pollard and Sag. These include coordinate structures and apposition.

4.1.3.1 Coordination

There is very little discussion by Pollard and Sag of coordinate structures. In Chapter Four of [Pollard and Sag, In press], they discuss the nature of the structure-sharing between the mother and the conjunct daughters, and argue in favor of the following version of the Coordination Principle (from p. 230):

(4.15) `Coordination Principle (weak version):`

```
In a coordinate structure, the CATEGORY and NONLOCAL value of
each conjunct daughter is subsumed by (is an extension of)
that of the mother.
```

However, they do not discuss the schemata required in order to generate coordinate structures. As coordinate structures are not headed, and the only schemata given in the book are for headed structures, none of these is applicable.

There are two distinct types of coordination in English, binary and iterative. Binary coordination joins exactly two items of the same category, marking both of them with an appropriate conjunction. Examples include "both A and B" and "either A or B." Iterative coordination joins two or more items of the same category, marking either only the last or all but the first with an appropriate conjunction. Examples include "A, B, C and D" and "A and B and C and

D." Instances of coordination may be embedded inside each other, giving rise to examples such as (4.16).[11]

(4.16) I've already invited Mutt and Jeff, Peter, Paul and Mary, Bob and Carol
and Ted and Alice, and Fred and both his wife and his mistress.

Binary coordination requires a new schema for non-headed structures, given in (4.17). This includes path equations which enforce the Coordination Principle given above.

(4.17)

```
define binarycoord:
phon2
context2
qip2
<x0 dtrs conjunct1> = <x1>
<x0 dtrs conjunct2> = <x2>
<x0 synsem local category> = <x1 synsem local category>
<x0 synsem local category> = <x2 synsem local category>
<x0 synsem local category head maj> = <x0 maj>
<x0 synsem local conj> = nil
<x1 synsem local conj> = both
<x2 synsem local conj> = and
<x0 synsem nonlocal> = <x1 synsem nonlocal>
<x0 synsem nonlocal> = <x2 synsem nonlocal>.
```

The current grammar has a separate schema for each set of conjunctions which may co-occur. The above rule for generating "both A and B" has different CONJ values from the rule for generating "either A or B." In future work, I will generalise these rules so that these co-occurrences are captured while still having only a single binary coordination schema in the grammar.

It is necessary to have a separate instantiation of the the binary coordination schema for each category that may be so coordinated. This is because the semantics of coordination is different for the different syntactic categories. The following is the rule for coordinating nominal categories.

---

[11] Examples such as these are highly ambiguous, and are disambiguated, if at all, by intonation and/or knowledge about the real world, such as which individuals belong together in a group. I do not discuss such resolution here; the parser currently finds all possible ambiguities.

(4.18)
```
n -> n n:
binarycoord
<x0 synsem local content index num> = plural
<x0 synsem local content restr first relation conj1> =
    <x1 synsem local content>
<x0 synsem local content restr first relation conj2> =
    <x2 synsem local content>
<x0 synsem local content restr first instance> =
    <x0 synsem local content index>
<x0 synsem local content restr rest> = end.
```

Again, future work will investigate whether it is possible to generalise across categories so that all of the binary coordination rules may be abstracted into a single, category-independent rule.

In addition, there must be a schema to license the local subtree whose mother is a substantive category with a non-null value for the CONJ feature, whose left daughter is a conjunction, and whose right daughter is of the same category as the mother, and has a null value for the CONJ feature. I treat this as an instance of the head-marker structure, allowing it to be licensed by Pollard and Sag's Schema 4 ([Pollard and Sag, In press], p. 47).

The schemata licensing iterative coordination structures are yet more complicated. For each syntactic category, there are two schemata necessary to cover the two types of iterative coordination. These differ primarily in their values for the CONJ feature. Rather than present the complete list of path equations for these, which are found in the Appendix A, I simply describe them, and then offer an example of each of the two kinds of iterative coordination.

UNICORN grammars do not have a means for representing zero or more or one or more categories. Therefore, I have encoded the rules for iterative coordination as a series of binary branching structures. In the first schema, the mother and both daughters are of the same category; the CONJ values of the mother and the lefthand daughter are `nil`, and that of the righthand daughter is `and`. In the second schema as well, the mother and both daughters are

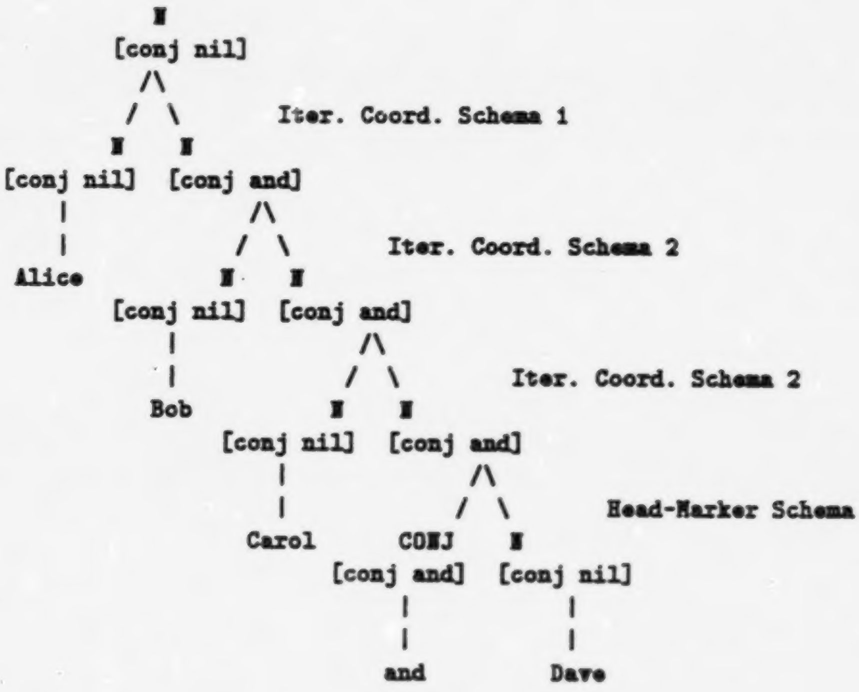
N
[conj nil]
Iter. Coord. Schema 1
N
[conj nil]
N
[conj and]
Alice
Iter. Coord. Schema 2
N
[conj nil]
N
[conj and]
Bob
Iter. Coord. Schema 2
N
[conj nil]
N
[conj and]
Carol
Head-Marker Schema
CONJ
[conj and]
N
[conj nil]
and
Dave

Figure 4.3: An example of iterative coordination with a single conjunction.

---

of the same category; the values of the mother and the righthand daughter are `and`, and that of the lefthand daughter is `nil`. Iterative coordination also uses the head-marker schema, just as binary coordination does.

The first type of iterated coordination is that which marks only the last element with a conjunction. This uses the first iterative coordination schema at the top, repeated tokens of the second throughout the middle of the structure, and the head-marker schema at the bottom. This is illustrated in the tree in Figure 4.3.

The second type of iterated coordination is that which marks all but the first conjunct with a conjunction. This alternates tokens of the first iterative coordination schema and the head-marker schema, with the former at the top and the latter at the bottom. This is illustrated in Figure 4.4.

4.2 The Knowledge Base

In this section, I discuss the implementation of a real-world knowledge base of a limited domain, and its integration into the rest of the understanding and acquisition system. The first subsection describes the prolog implementation of the knowledge base itself, while the second briefly raises the issue of non-literal meaning. The third section describes how semantic checking is implemented in the system. The fourth discusses the issue of Montagovian vs. Davidsonian semantic representation, which arises in the grammar as a result of the use of real-world knowledge for disambiguation via semantic checking.

4.2.1 Real-World Knowledge of a Limited Domain

This section discusses the real-world knowledge base used with the system. This knowledge base is domain-specific, containing the information necessary to properly deal with sentences from actual texts pertaining to driving actions and traffic accident reports. The interface of the knowledge base with the grammar processor is discussed, and it is shown how the knowledge base is interleaved with the parsing process, for disambiguation and the elimination of unpromising candidate parses.

Information in the real-world knowledge base is of two basic types. One tells what entities can fill a given role of a script. The other specifies what properties a given entity may have. These are encoded in the prolog predicates `can_be_role_of` and `can_modify`, respectively.

The predicate `can_be_role_of` takes three arguments: the filler, the role and the predicate. For example, the prolog statements in (4.19) assert that entities of the types "frame," "photograph" and "picture" can fill the role of "picture" in the predicate "takepicture."

(4.19)
```
can_be_role_of(frame,picture,takepicture).
can_be_role_of(photograph,picture,takepicture).
can_be_role_of(picture,picture,takepicture).
```

Statements such as this are unusual. The "picture" role of the "takepicture" predicate is very restricted, and can only be filled by a few entities. These may be listed, each in a separate prolog clause.

```
           N
        [conj nil]
          /\
         /  \       Iter. Coord. Schema 1
         N   N
[conj nil]  [conj and]
    |             /\
    |            /  \       Head-Marker Schema
 Alice       CONJ    N
        [conj and]  [conj nil]
            |             /\
            |            /  \         Iter. Coord. Schema 1
           and           N   N
                [conj nil]  [conj and]
                    |             /\
                    |            /  \        Head-Marker Schema
                   Bob       CONJ    N
                        [conj and]  [conj nil]
                            |             /\
                            |            /  \       Iter. Coord. Schema 1
                           and           N   N
                                [conj nil]  [conj and]
                                    |             /\
                                    |            /  \       Head-Marker Schema
                                  Carol      CONJ    N
                                        [conj and]  [conj nil]
                                            |           |
                                            |           |
                                           and         Dave
```

Figure 4.4: An example of iterative coordination with multiple conjunctions.

More often, the possible fillers of a role fall into a natural class, whose membership may be very large, or possibly even infinite. For example, the set of entities that can fill the "thinker" role of the "think" relation may be roughly described as the class of humans.[12] Because the restricted domain does not contain an infinite number of entities, it would theoretically be possible to list all of the humans in the domain, as was done for (4.19). However, this fails to capture the fact that these listed entities form a natural class. This is done by filling the role with a general type, as in (4.20).

```
(4.20)   can_be_role_of(Arg,thinker,think) :-
               is_person(Arg).
```

Further prolog clauses in the knowledge base specify the ways in which an entity can be known to be a person. One way is to be an entity which is a subtype of the entity-type `person`, and the other is to be known to be the name of a person.

```
(4.21)   is_person(claimant).
         is_person(driver).
         is_person(insured).
         is_person(owner).
         is_person(party).
         is_person(people).
         is_person(person).
         is_person(passenger).
         is_person(police).
         is_person(witness).
         is_person(Arg) :-
               is_name_of(Arg,person).

(4.22)   is_name_of(edwardladewick,person).
         is_name_of(meuser,person).
         is_name_of(paulameuser,person).
```

---

[12] It could be argued whether this is actually the right class to fill the role of "thinker" in the "think" relation, but this is sufficient for expository purposes.

```
is_name_of(wordenmartin,person).
is_name_of(officergeorgewissmiller,person).
is_name_of(num1,person).
is_name_of(num2,person).
is_name_of(num3,person).
```

Statements such as these are used to create a semantic hierarchy. An entity may be declared to be an instance of a node at any level of the hierarchy, specifying that the entity is of a certain semantic type. There may also be a number of intermediate levels, like those of an ISA hierarchy, so that an entity is specified to be of a certain type, while elements of that type are all members of a higher type, and so on.

The `can_modify` predicate takes two arguments, a quality and an entity of which the quality may be predicated. This predicate makes similar use of the semantic hierarchy. It can take atomic values, as in (4.23), to denote a fixed expression. It can also make reference to the same hierarchical levels as `can_be_role_of`, as in (4.24).

```
(4.23)    can_modify(rearview,mirror)

(4.24)    can_modify(belligerent,Arg) :-
                is_person(Arg).
```

Not just entities, but also properties participate in the semantic hierarchy in this way. For example, the clauses in (4.25) assert that a person, street or vehicle can have any quality which is a directional movement, while (4.26) contains the list of such qualities.

```
(4.25)    can_modify(Qual,Arg) :-
                is_moving_direction(Qual),
                is_person(Arg).
          can_modify(Qual,Arg) :-
                is_moving_direction(Qual),
                is_street(Arg).
          can_modify(Qual,Arg) :-
                is_moving_direction(Qual),
                is_vehicle(Arg).
```

(4.26)
```
is_moving_direction(northbound).
is_moving_direction(southbound).
is_moving_direction(eastbound).
is_moving_direction(westbound).
```

This licenses the use of these words in contexts such as the following.

(4.27) a. Meuser was eastbound on Elm.

b. The driver skidded into the southbound lane.

c. She was attempting to turn right onto Elm, which is northbound until it crosses the railroad tracks, and then heads northwest.

d. The collision propelled the driver into the path of the southbound traffic.

e. There was a westbound pick-up truck headed directly at the driver of Vehicle Number One.

4.2.2 Non-Literal Meaning

For the most part, the knowledge base has been formulated in such a way that only literal utterances are deemed semantically acceptable. However, as is well-known, the use of metaphor and other non-literal types of meaning is not a rare and isolated phenomenon, but pervades all natural language usage.

Chapter Eight gives a more detailed account of how such non-literal meanings may be handled in a unification-based theory of grammar. For now, I merely note which types of non-literal meaning occur most frequently in the given corpus, and incorporate a brute-force solution to these in the current knowledge base.

The most common type of non-literal meaning in the corpus is metonymy, the mention of one entity to refer to some other entity related to it in a systematic way. In the given corpus, this is most often done with references to vehicles and their drivers. The metonymy works in either direction; a vehicle can be mentioned in order to refer to the driver of that vehicle, or the driver can be named to denote the vehicle they are driving. Examples of both types are given in (4.28).

(4.28) a. The pick-up did not see the semitrailer truck.

87

b. The Cadillac stated that the other vehicle was stopped at a stop sign.

c. I was hit in the front right side by the Buick.

d. The first impact severely reshaped Meuser's front bumper.

The desired analysis of this phenomenon would be to formulate two referring functions, licensing the denotation of a vehicle by a reference to the driver of that vehicle, and vice versa. However, I have not implemented referring functions in the current grammar. Therefore, a workable intermediate solution is to stipulate (falsely) in the knowledge base that vehicles are persons, and persons are vehicles. This is the solution I have implemented in order to allow the particular non-literal utterances found in the given corpus.

It should be clear that this is only a temporary solution, and will not be generally applicable or extensible. The two referring functions are used felicitously only in a few restricted environments, such as those of driving and traffic accidents. Outside of such domains, it is not reasonable to refer to a person by mentioning the vehicle they drive, or vice versa. Thus, outside of specialized contexts, the following sentences are not likely to be understood as employing these referring functions.

(4.29) a. The pick-up truck at Table Six wants some ketchup for his eggs.

b. The Cadillac complained that the other vehicle had stepped all over her
toes during the Viennese waltz.

c. The boxer hit me hard, knocking out my front driver's side window.

d. The impact of his wife's death severely reshaped Meuser's front bumper.

By encoding these falsehoods in the knowledge base, I have effectively allowed these two referring functions to apply without restrictions. However, for the given corpus, there are no instances where an anomalous reading is allowed by the unrestricted application of these referring functions. This is therefore an acceptable solution, until the use of referring functions is actually implemented in the grammar.

4.2.3 Semantic Checking

Semantic checking is the process of consulting the knowledge base to determine whether a candidate rule to be used in parsing would produce a semantically allowable combination. It is interleaved with parsing, and thus can eliminate semantically anomalous candidates before a great deal of computational effort has been expended.

Semantic checking is invoked during the completion step of Earley's algorithm. At this step, a certain Phrase Structure rule is being considered during the course of the parse, and all of the elements on the right-hand side of this rule have been found.

However, before the rule is actually accepted as a possible completion, the semantic checker is invoked. This retrieves the entity type from the semantic representation of each of the daughters, and determines whether the rule is one in which one daughter modifies the other, or acts as the argument of the other. If the relation between the daughters is neither modifier-modified nor functor-argument, then the rule is declared to be semantically acceptable by default. This is because the other constructions licensed by rules of the grammar are those in which one of the daughters has no semantic content, and therefore cannot generate a semantic anomaly. This includes head-marker and head-specifier constructions.

4.2.4 An Issue of Semantic Representation

In this section, I discuss two majors approaches to semantic representation. Each of these has been developed by a number of researchers in linguistics, philosophy and computer science, but I refer to them by the names of the authors with whom they have come to be primarily associated, Montague and Davidson. While each has advantages and disadvantages, I argue that for the purposes of the present work, a Davidsonian representation is more appropriate.

4.2.4.1 Montague Semantics

In a Montagovian approach to semantics, every syntactic rule has an associated semantic rule, specifying which daughter is the function (or predicate) and which the argument. This licenses a hierarchical semantic "tree," often isomorphic with the syntactic tree, representing function application. For example, a feature structure representation of (4.30) would be as in (4.31).

(4.30) John parked the car at the corner.

$$(4.31)\quad \begin{bmatrix} \texttt{Argument:} & \texttt{John} \\ \texttt{Predicate:} & \begin{bmatrix} \texttt{Argument:} & \begin{bmatrix} \texttt{Argument:} & \begin{bmatrix} \texttt{Argument:} & \texttt{car} \\ \texttt{Predicate:} & \texttt{the} \end{bmatrix} \\ \texttt{Predicate:} & \texttt{parked} \end{bmatrix} \\ \texttt{Predicate:} & \begin{bmatrix} \texttt{Argument:} & \begin{bmatrix} \texttt{Argument:} & \texttt{corner} \\ \texttt{Predicate:} & \texttt{the} \end{bmatrix} \\ \texttt{Predicate:} & \texttt{at} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

The problem with this approach becomes apparent when we consider how to implement the kind of semantic checking described earlier in this section. For example, suppose the knowledge base is to be consulted to determine whether the entity denoted by "John" is the kind of thing which can park a car.

The most straightforward way to ask this question in the Montagovian approach would be to determine whether "John" can be the argument of the "park" predicate, or of a complex predicate denoting "park the car." However, in this example, neither of these is available as a predicate of which "John" is the argument. In fact, the feature structure representing "park the car" is not a predicate at all, but an argument of a higher predicate.

In order to get the information needed for semantic checking, it is necessary to perform intricate manipulations on complex predicates and arguments. Moreover, there is no principled way to determine ahead of time where in the feature structure the desired information will be.

4.2.4.2 Davidsonian Semantics

Davidsonian representation uses a script-like semantics. Here, function categories, such as verbs, determine which script is active, while argument categories fill in the slots of the active script. In many such representation, some slots, such as those representing modifiers, may be optional. This produces a semantic structure which is more 'flat.'

$$(4.32)\qquad \begin{bmatrix} \texttt{Relation:} & \texttt{parked} \\ \texttt{Parker:} & \texttt{John} \\ \texttt{Parked:} & \begin{bmatrix} \texttt{Type:} & \texttt{car} \\ \texttt{Def:} & \texttt{the} \end{bmatrix} \\ \texttt{Location:} & \begin{bmatrix} \texttt{Type:} & \texttt{corner} \\ \texttt{Def:} & \texttt{the} \end{bmatrix} \end{bmatrix}$$

Here the information needed for semantic checking is more accessible. It is now straightforward to ask questions such as the following:

- Can `John` fill the role of `Parker` in a `Park` relation?
- Can something whose entity-type is `car` fill the `Parked` role in a `Park` relation?

One possible drawback of Davidsonian representation is that it relates the various arguments of a function only to that function, not to each other. For example, it is only possible to ask whether something can fill a given slot of a certain script, not whether it can fill that slot when the script has another slot filled in a particular way. However, this has not been a limiting factor in the knowledge checking currently being done with the system.

Chapter 5

The Semantics of Adverbs and Verbs in HPSG

In this chapter, I argue for a modification to the feature structure given in [Pollard and Sag, In press] for the semantic representation of verbs and verb phrases in HPSG. This is motivated by concerns of efficiency in the use of world knowledge for semantic checking. In particular, I argue that the CONTENT value for a verb must take as its value not simply a parameterised state of affairs (PSOA), but a structure which includes both a PSOA and a set of restrictions. As a consequence, the semantic feature structure of verbs is parallel to that of common nouns and noun phrases, which have separate feature paths for an INDEX and a set of restrictions.

This modification is motivated by an exploration of the question of what the CONTENT value for adverbs in HPSG should be, in particular those adverbs which modify verb phrases. One possible initial hypothesis would be that the CONTENT of such adverbs should look much like that of adjectives. I show that this cannot be the case given the feature structure system of [Pollard and Sag, In press], because of differences between the semantic feature structure of verb phrases and that of common noun phrases. After examining other alternatives for adverbial semantics, I return to this initial hypothesis, and show that it can easily be realized if the proposed modification is made to the feature structure system for the semantics of verbs.

There are three other major approaches which I examine. The first is to postulate a slot for every type of adverbial modifier. It is fairly easy to then write grammar rules that license the proper structures. However, there are practical reasons for avoiding this approach, such as the difficulty in finding the complete set of such slots, and the impossibility of allowing more than one modifier of a given slot type with this approach.

The second approach is to treat VP-modifying adverbs semantically as raising verbs. That is, the semantic content of the adverbial modifier is the relation which has the widest semantic scope in the VP, taking the semantic content of the modified VP as an argument. This approach is in the spirit of Montague semantics, and is discussed in [Carpenter, 1989]. I show that while the semantic of both the modifiers and the PSOA may be captured with this approach, the resulting structure is difficult or impossible to use in checking the semantic well-formedness of a structure against a database of world knowledge. I argue that this is not merely an implementation problem, but reveals the counter-intuitiveness of the proposed semantic structure.

The third approach is to formulate the semantics of adverbials so that each specifies an element on a set of restrictions to the semantic content of the modified verb phrase, as adjectives specify restrictions on the semantic content of noun phrases. Two possible modifications to the feature structure of verb phrases are presented to license structures that reflect the desired semantics. However, the first, which uses the mechanism of difference lists, requires a bookkeeping feature slot outside of CONTENT, whose only purpose is to manipulate the CONTENT feature in the desired way.

Finally, I propose a second type of modification to the feature structure of verb phrase semantics, which avoids the weaknesses of the other approaches. I show that by positing a feature PSOA in the semantic CONTENT of the VP, paralleling INDEX in NP semantics, we can encode the specification of restriction sets in a straightforward way. I then briefly discuss the implications of this modification for the semantic representation of other parts of speech.

5.1 The Initial Hypothesis: Adverbs vs. Adjectives

There is no explicit treatment of VP-modifying adverbs in [Pollard and Sag, In press]. However, like adjectives, adverbs are semantic modifiers and syntactic adjuncts. Therefore, it is reasonable to explore the possibility that the semantic feature structure of adverbs is similar to adjectives. A detailed treatment of at least a certain class of adjectives is given in Chapter One of [Pollard and Sag, In press], in Section 1.9 on adjuncts.

The adjectives discussed here are of the class whose semantics provides something like an intersection relation on the modified element. For example, part of the semantics of "ball" is that it refers to something that is restricted to have the semantic property of being a ball. This is encoded in the feature structure of HPSG by saying that the CONTENT feature has a value which includes a restriction (an element of the RESTR set) which says that the entity fills the INSTANCE slot of a FS whose RELATION is 'ball'. This is illustrated in the Attribute Matrix Value representation for "ball," given in (5.1) (adapted from Pollard and Sag's (45), on p. 44).

(5.1)
$$\begin{bmatrix} \text{CATEGORY} & \begin{bmatrix} \text{HEAD} & \text{noun} \\ \text{SUBCAT} & \langle \text{DetP} \rangle \end{bmatrix} \\ \text{CONTENT} & \begin{bmatrix} \text{INDEX} & \boxed{1} \\ \text{RESTR} & \left\{ \begin{bmatrix} \text{RELN} & \text{ball} \\ \text{INST} & \boxed{1} \end{bmatrix} \right\} \end{bmatrix} \end{bmatrix}$$

Part of the semantics of "red" is a set of restrictions, which is the union of the restriction set of the element it modifies and the additional restriction that the referent must fill the INSTANCE slot of the relation RED. The lexical entry for "red," is given in Pollard and Sag's (58) of Chapter One (p. 52), repeated here as (5.2). The semantic content of a modified noun is the index of the noun, and the union of the sets of restrictions specified in the lexical entries of the noun and the modifiers, as shown in the AVM of "red ball," given in (5.3).

(5.2)
$$\begin{bmatrix} \text{CATEGORY} & \begin{bmatrix} \text{HEAD} & {}_{\text{adj}}\begin{bmatrix} \text{MOD} & \text{N}' : \begin{bmatrix} \text{INDEX} & \boxed{1} \\ \text{RESTR} & \boxed{2} \end{bmatrix} \\ \text{PRD} & - \end{bmatrix} \\ \text{SUBCAT} & \langle\rangle \end{bmatrix} \\ \text{CONTENT} & \begin{bmatrix} \text{INDEX} & \boxed{1} \\ \text{RESTR} & \left\{ \begin{bmatrix} \text{RELN} & \text{red} \\ \text{ARG} & \boxed{1} \end{bmatrix} \right\} \cup \boxed{2} \end{bmatrix} \end{bmatrix}$$

$$(5.3)\quad \left[\begin{array}{ll} \text{CATEGORY} & \left[\begin{array}{ll} \text{HEAD} & \text{noun} \\ \text{SUBCAT} & \langle \text{DetP} \rangle \end{array}\right] \\ \text{CONTENT} & \left[\begin{array}{ll} \text{INDEX} & \boxed{1} \\ \text{RESTR} & \left\{ \left[\begin{array}{ll} \text{RELN} & \text{ball} \\ \text{INST} & \boxed{1} \end{array}\right], \left[\begin{array}{ll} \text{RELN} & \text{red} \\ \text{ARG} & \boxed{1} \end{array}\right] \right\} \end{array}\right] \end{array}\right]$$

The parallel between adjectives and adverbs is intuitive and appealing. Just as the "red" in "red ball" restricts the set of balls that can be referred to by the noun phrase, in the same way "quickly" restricts the set of actions that can be referred to by the verb phrase "walks quickly." And just as multiple adjectives (including adjectival prepositional phrases) further specify the set of restrictions on the NP's referent, as in "misshapen little red ball with a hole in it," so multiple adverbial phrases further specify the restrictions on a VP's referent, as in "walks quickly across the bridge with a smile on his face."

One straightforward treatment of adverbs, therefore, would be to say that they are exactly like adjectives, except that they modify an element whose major category is V (verb) rather than N (noun). However this does not work, given the semantic representations of [Pollard and Sag, In press]. In the lexical entry for "red," seen in (5.2), the restriction set in the CONTENT of the adjective contains a reentrancy with the restriction set of the CONTENT of the noun which it modifies.

A parallel lexical entry is not possible for adverbs, because unlike common nouns, the semantic CONTENT of verbs does not contain the features INDEX and RESTR. Rather, the CONTENT of a verb is parameterized state of affairs (PSOA), which consists of a relation and a set of participants. For example, the lexical entry for "walks," Pollard and Sag's (8) of Chapter One is given as (5.4).

(5.4)
$$\begin{bmatrix} \mathtt{PHON} & \mathtt{walks} \\ \mathtt{SYNSEM|LOC} & \begin{bmatrix} \mathtt{CATEGORY} & \begin{bmatrix} \mathtt{HEAD} & \mathtt{verb[fin]} \\ \mathtt{SUBCAT} & < \mathtt{NP[nom]}_{\boxed{1}\,\mathtt{[3rd,sing]}} > \end{bmatrix} \\ \mathtt{CONTENT} & \begin{bmatrix} \mathtt{RELN} & \mathtt{walk} \\ \mathtt{WALKER} & \boxed{1} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

The structure in (5.4) contains no INDEX for the adverbial semantic content to unify against, and no restriction set for the adverb further specify. At first, this may not seem like an obstacle at all, since the mechanism of unification is available for relating constituents and their daughters. According to the Semantics Principle, the CONTENT value of the mother in a head-adjunct structure is token-identical to that of the adjunct daughter. The rule licensing VP-modifying adverbs is an instantiation of Schema 5, shown in 5.5 (from [Pollard and Sag, In press], p. 53), and is therefore subject to this principle.

(5.5) `(Schema 5) a phrase with DTRS value of sort` *head-adjunct-structure* `(`*head-adj-struc*`), such that the MOD value fo the adjunct daughter is token-identical to the SYNSEM value of the head daughter.`

Therefore, the lexical entry of the adverb should state that its CONTENT unifies with that of the element it modifies. Since the verb has no restriction, the restriction set on the mother is simply that of the adjunct daughter, and the rest of the mother's CONTENT is that of the head daughter. Thus as a first guess, we might formulate the lexical entry for "quickly" as in (5.6).

(5.6)

$$\begin{bmatrix} \texttt{CATEGORY} & \begin{bmatrix} \texttt{HEAD} & {}_{\texttt{adv}}\begin{bmatrix} \texttt{MOD} & \texttt{V}:\boxed{2}\begin{bmatrix}\texttt{RELN} & \boxed{1}\\ \vdots & \end{bmatrix} \\ \texttt{PRD} & - \end{bmatrix} \\ \texttt{SUBCAT} & <> \end{bmatrix} \\ \texttt{CONTENT} & \begin{bmatrix} \texttt{RESTR} & \left\{\begin{bmatrix}\texttt{RELN} & \texttt{quickly}\\ \texttt{ARG} & \boxed{1}\end{bmatrix}\right\} \cup \boxed{2} \end{bmatrix} \end{bmatrix}$$

This captures the intuition that adverbial modifiers provide restrictions on the referent of the verb. In fact, this would work for the above case, producing a structure for "walks quickly" as in (5.7).[1] The CONTENT value of the mother is token-identical to that of the adjunct daughter, which consists of the CONTENT of the element it modifies, plus the added restriction of 'quickly.'

(5.7) walks quickly

$$\begin{bmatrix} \texttt{CATEGORY} & \begin{bmatrix} \texttt{HEAD} & \texttt{verb[fin]} \\ \texttt{SUBCAT} & < \texttt{NP[nom]}_{\boxed{1}\texttt{[3rd,sing]}} > \end{bmatrix} \\ \texttt{CONTENT} & \begin{bmatrix} \texttt{RELN} & \boxed{2}\texttt{walk} \\ \texttt{WALKER} & \boxed{1} \\ \texttt{RESTR} & \left\{\begin{bmatrix}\texttt{RELN} & \texttt{quickly}\\ \texttt{ARG} & \boxed{2}\end{bmatrix}\right\} \end{bmatrix} \end{bmatrix}$$

The difficulty comes when there is more than one adverbial modifier, as in "walks quickly here."[2] This phrase should have two daughters: one which is identical to (5.7), and the other given in (5.8), the lexical entry for "here," which is identical, *ceteris paribus*, to that for "quickly."

---

[1] A possible objection to this approach might be that it requires the unification of two different kinds of objects. Something whose CONTENT value has the slots INDEX and RESTRICTION, such as a common noun, is of the type ref-object, while a relation and its participants is a parameterised state of affairs (PSOA). The desired solution to this difficulty would be to allow a feature slot of type RESTRICTION to occur in an object of type ref-object. This could be done either by stipulating that any PSOA may have a slot for RESTR, or by setting up a supertype, which results from the unification of a PSOA with a restriction list.

[2] While this example sounds stilted, I want to look only at single-word adverbials for now, to avoid the complications that arise from adverbial prepositional phrases.

(5.8)

$$\begin{bmatrix} \text{CATEGORY} & \begin{bmatrix} \text{HEAD} & {}_{\text{adv}}\begin{bmatrix} \text{MOD} & \text{V}:\boxed{2}\begin{bmatrix}\text{RELN} & \boxed{1}\\ \vdots & \end{bmatrix} \\ \text{PRD} & - \end{bmatrix} \\ \text{SUBCAT} & <> \end{bmatrix} \\ \text{CONTENT} & \boxed{2}\begin{bmatrix} \text{RELN} & \boxed{1} \\ \vdots & \\ \text{RESTR} & \left\{\begin{bmatrix}\text{RELN} & \text{here}\\ \text{ARG} & \boxed{1}\end{bmatrix}\right\}\end{bmatrix}\end{bmatrix}$$

However, the AVMs given in (5.7) and (5.8) fail to qualify as the daughters of a head-adjunct structure under any instantiation of Schema 5. They give different values for the restriction set on the PSOA, so the union of the restriction relation 'here' with the CONTENT of (5.7) fails to produce a well-formed structure.

The problem is that the adverb's CONTENT must be the same as that of the VP it modifies, *except* for the RESTR value. The RESTR value of the modifier must be the same as that of the element it modifies, except that the new restriction contributed by the adverbial itself is added to the set. However, it is not possible for a unification operation to specify that one feature structure is unified with another structure, except that a given value is to be ignored or modified. Therefore, the CONTENT values of (5.7) and (5.8) simply fail to unify. This eliminates the possibility of giving adverbs a semantic structure similar to that of adjectives.

5.2 Assigning Sorts to Adverbial Modifiers

As the semantic content of verbs in HPSG uses a Davidsonian, or script-like semantic representation, a possible solution to the problem outlined above might be to extend the slot-filler type of semantics to adverbial modifiers. That is, rather than representing the semantics of adverbials as restrictions on the semantics of the VPs they modify, each adverbial should have as its semantic content a slot which it fills, and the value with which it fills that slot. For example, the underlined adverbial modifiers in the following examples would have the slot-filler values given in parentheses. The feature structure of right in (5.9a) is given in (5.12).

(5.9) a. Then you will turn right. (direction : right)

b. The vehicle veered off to the right. (direction : right)

(5.10) a. You'll stay on this road forever (extent : forever).

b. Keep going straight for two miles. (extent : two miles)

(5.11) Take a right onto Elm Street. (path-change : Elm Street)

(5.12)

$$\begin{bmatrix} \text{CATEGORY} & \begin{bmatrix} \text{HEAD} & {}_{\text{adv}}\begin{bmatrix} \text{MOD} & \text{V}:\boxed{2}\begin{bmatrix} \text{RELN} & \boxed{1} \\ \vdots & \end{bmatrix} \\ \text{PRD} & - \end{bmatrix} \\ \text{SUBCAT} & \langle\rangle \end{bmatrix} \\ \text{CONTENT} & \boxed{2}\begin{bmatrix} \text{RELN} & \boxed{1} \\ \vdots & \\ \text{DIR} & \text{right} \end{bmatrix} \end{bmatrix}$$

This approach works well for certain classes of adverbials. It is certainly the representation most compatible with a script-like semantic representation for verbs whose subcategorization frame specifies an adverbial, such as "put" or "head." The subcategorization lists of these verbs must mention a phrase whose type is "location" and "direction" respectively.[3]

This also seems to be an attractive approach for adverbials headed by so-called case-marking prepositions. Here the preposition has no semantic content, but only marks the semantic role (corresponding to Fillmore's notion of case) of its object. Perhaps the best examples of case-marking prepositions are the "to" of dative constructions and the "by" of passives, shown in

---

[3] I here treat these arguments as syntactically adverbials. Thus, there is no separate category of P or PP in the grammar, with all prepositions treated as transitive adjectives or adverbs. The arguments given here for modifications to the semantic feature structure system could all be formulated equally well in an analysis which includes Preposition as a syntactic category.

(5.13) and (5.14).

(5.13) John gave a bone to Fido.

(5.14) Fido was taken to the vet by John.

In practice, however, it can be difficult to decide whether a given preposition is case-marking or contentful. The preposition "around" in (5.15) does have semantic content, specifying a relation between its object and the element which the whole prepositional phrase modifies. It could be argued that "at" in (5.16) has no semantic content except to mark its object as denoting a location.

(5.15) John took three laps around the house.

(5.16) Turn right onto Green Street at the corner.

Assigning types to adverbial modifiers does make some correct predictions. This approach predicts that no two adverbial modifiers of the same type can occur modifying a single VP. This is at least correct for a large number of proposed adverbial types, as the following examples show (though I discuss exceptions below).[4]

(5.17) * Turn right at this next corner at the intersection of Green and Lincoln.

(5.18) * From this intersection, head north left.

(5.19) * Keep going straight for seven blocks for five minutes.

Although there might at first appear to be a one-to-one correspondence between prepositions and adverbial types, this is not the case. This approach correctly predicts that while a VP may not be modified by two prepositional phrases headed by the same preposition if they are of the same type, as in (5.17), it is perfectly all right to have two prepositional phrases headed by the same preposition if they are of different types, as in (5.20).

(5.20) Meet me at the mall at 4:00.

There are two problems with this approach. The first is coming up with a complete inventory of the types of adverbials required. As we have just seen, there is not a simple one-to-one correspondence between adverbial types and prepositions. As with Fillmorian semantic cases, there are some adverbial types which are clear, but others that are subtle, and formulating an exhaustive list is an elusive goal. Any attempt to come up with a complete list of adverbial types will probably need to have one or more "garbage-can" types, to catch all of those that do not fit any other category. "Manner" is a likely candidate for this.

---

[4] There is a grammatical sentence very much like (5.17), with a comma between the two adverbials, which marks them as appositional. In the ungrammatical sentence, both adverbials modify the VP without forming a constituent themselves.

The second problem, related to the first, is that for some adverbial types, it does seem to be possible to have more than one modifier of a given type. As a rough rule, this becomes more possible for types that more closely resemble "garbage-can" categories, and is especially true of manner adverbs.

(5.21) John came quietly into the room with an embarrassed grin on his face.

In (5.21), the underlined adverbials are not appositional, yet there is no obvious adverbial type to which one but not the other should belong.

5.3 Attaching Restrictions to Verb Phrase Semantics

While the assignment of types may well be appropriate for some adverbials, we have seen that there are others, such as PPs headed by contentful prepositions, that are best described as restricting the content of the VP in the same way that adjectives restrict the content of the NP. In this section, I investigate the approach of representing such restrictions directly in the CONTENT value of the VP.

I first show a modification to the feature structure that would allow the restrictions to be represented inside the CONTENT of the VP, as a slot which co-occurs with every RELATION. This requires a slot external to the CONTENT in order to implement difference lists. I then propose an alternative in which everything that is currently represented as the CONTENT of a VP is placed inside a PSOA feature, and a RESTR feature is added to the CONTENT of VPs, just as it is for NPs.

5.3.1 The Difference List Approach

It was shown in Section 5.2 that it is not possible for the CONTENT of an adverbial to unify with that of the element it modifies while adding a RESTR slot, because this disallows more than a single adverbial on a given VP. If we want to keep the restrictions contributed by the adverbials within the semantic content of the VP, then, what we need is for the value of the RESTR within CONTENT to be a list. That is, the CONTENT of "walks quickly here" should be as in (5.22).

(5.22) **walks quickly here**

$$\begin{bmatrix} \text{CONTENT} & \begin{bmatrix} \text{RELN} & \text{walk} \\ \text{WALKER} & \langle\ \text{SYNSEM LOCAL CATEGORY SUBCAT FIRST} \\ & \text{SYNSEM LOCAL CONTENT INDEX}\ \rangle \\ \text{RESTR} & \begin{bmatrix} \text{FIRST} & \text{quickly} \\ \text{REST} & \begin{bmatrix} \text{FIRST} & \text{here} \\ \text{REST} & \text{end} \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Let us now see what the CONTENT of an adverbial must be in order to produce such a VP CONTENT. The adverb's CONTENT must be the same as that of the VP it modifies, except for the RESTR value. The RESTR value must be the same as that of the element it modifies, except that the new restriction contributed by the adverbial itself is added either to the beginning or the end of the list.

It is not possible for a unification operation to specify that one feature structure is unified with another structure, except that a given value is to be ignored or modified.[5] If the adverbial is to share the CONTENT of the element it modifies, it must share the whole thing, including the RESTR slot. What is needed, then, is a way to leave the RESTR list of the VP always partially specified, so that any number of adverbials can add to it. Difference lists provide the necessary mechanism for doing this.

I will not attempt to explain the theory behind difference lists, but an example should provide the motivation and show how the implementation works. The feature structure in (5.23) shows the desired semantic CONTENT of the VP "walks quickly here," and the semantic CONTENT that its constituent elements must have in order to produce the correct structure.

---

[5] This would be possible if the CONTENT feature always had the same set of feature slots. In this case, we could simply list all of the feature slots except RESTR, and say that the value of the adverbial for each of these is the same as that of the element it modifies. This approach is not available for adverbials, however, because the CONTENT of verbs is lexically determined. Each verb has a lexically specified RELN, and the number and type of the other slots are determined by the RELN.

BEST COPY AVAILABLE

(5.23) walks

$$\begin{bmatrix} \text{CONTENT} & \boxed{1} \begin{bmatrix} \text{RELN} & \text{walk} \\ \text{WALKER} & \langle \text{SYNSEM LOCAL CATEGORY SUBCAT FIRST} \\ & \text{SYNSEM LOCAL CONTENT INDEX} \rangle \\ \text{RESTR} & \boxed{2}\ \text{T} \end{bmatrix} \end{bmatrix}$$

(5.24) quickly

$$\begin{bmatrix} \text{SUBCAT} & \langle \text{VP}_{\boxed{1}} \rangle \\ \text{CONTENT} & \boxed{1}\ \text{RESTR}\ \boxed{2} \begin{bmatrix} \text{FIRST} & \text{quickly} \\ \text{REST} & \text{T} \end{bmatrix} \end{bmatrix}$$

(5.25) walks quickly

$$\begin{bmatrix} \text{CONTENT} & \boxed{1} \begin{bmatrix} \text{RELN} & \text{walk} \\ \text{WALKER} & \langle \text{SYNSEM LOCAL CATEGORY SUBCAT FIRST} \\ & \text{SYNSEM LOCAL CONTENT INDEX} \rangle \\ \text{RESTR} & \boxed{2} \begin{bmatrix} \text{FIRST} & \text{quickly} \\ \text{REST} & \boxed{3}\ \text{T} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

(5.26) here

$$\begin{bmatrix} \text{SUBCAT} & \langle \text{VP}_{\boxed{1}} \rangle \\ \text{CONTENT} & \boxed{1}\ \text{RESTR}\ \boxed{3} \begin{bmatrix} \text{FIRST} & \text{T} \\ \text{REST} & \begin{bmatrix} \text{FIRST} & \text{here} \\ \text{REST} & \text{end} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

(5.27) **walks quickly here**

$$\left[\texttt{CONTENT}\ \boxed{1}\left[\begin{array}{ll} \texttt{RELN} & \texttt{walk} \\ \texttt{WALKER} & <\texttt{SYNSEM LOCAL CATEGORY SUBCAT FIRST} \\ & \texttt{SYNSEM LOCAL CONTENT INDEX}> \\ \texttt{RESTR} & \boxed{2}\left[\begin{array}{ll} \texttt{FIRST} & \texttt{quickly} \\ \texttt{REST} & \boxed{3}\left[\begin{array}{ll} \texttt{FIRST} & \texttt{here} \\ \texttt{REST} & \texttt{end} \end{array}\right] \end{array}\right] \end{array}\right]\right]$$

As stated in Schema 5 of [Pollard and Sag, In press], the CONTENT of the mother at each level of the tree is token-identical with that of its adjunct daughter. The CONTENT of each adverb is lexically specified to be token-identical with that of the element it modifies, so by transitivity, the CONTENT of all of the V nodes are also token-identical. So the restriction lists of the V nodes at each level of the tree are token-identical, but the amount of information that is specified for them differs according to how many adverbials have made their semantic contribution at that level.

The question then is how to incorporate the semantic contribution of the adverbial at each level, to augment the information specified on the V node at each level. We want to say that the lexical entry for "quickly" specifies that the <CONTENT RESTR FIRST> value of its sister (and hence of its mother as well) is QUICKLY. But then, we would have to say that the lexical entry for "here" specifies that the <CONTENT RESTR REST FIRST> value of its sister is "here." For any number of adverbials in a phrase, there would have to be an increasing number of RESTs to reach the end of the list.

What is needed is a place-holder for the end of the list that is itself external to the restriction list. Let us call it EOCRL, for END-OF-CONTENT-RESTRICTION-LIST. We can then give the complete semantic CONTENT for the above elements, as in (5.28).

(5.28) **walks**

$$\begin{bmatrix} \text{CONTENT} & \boxed{1}\begin{bmatrix} \text{RELN} & \text{walk} \\ \text{WALKER} & <\text{SYNSEM LOCAL CATEGORY SUBCAT FIRST} \\ & \text{SYNSEM LOCAL CONTENT INDEX}> \\ \text{RESTR} & \boxed{2}\ \text{T} \end{bmatrix} \\ \text{EOCRL} & \boxed{2} \end{bmatrix}$$

(5.29) **quickly**

$$\begin{bmatrix} \text{SUBCAT} & <\text{VP}_{\boxed{1}}> \\ \text{CONTENT} & \boxed{1}\ \text{RESTR}\ \boxed{3} \\ \text{EOCRL} & <\text{quickly}, \ldots> \end{bmatrix}$$

(5.30) **walks quickly**

$$\begin{bmatrix} \text{CONTENT} & \boxed{1}\begin{bmatrix} \text{RELN} & \text{walk} \\ \text{WALKER} & <\text{SYNSEM LOCAL CATEGORY SUBCAT FIRST} \\ & \text{SYNSEM LOCAL CONTENT INDEX}> \\ \text{RESTR} & \boxed{2}\begin{bmatrix} \text{FIRST} & \text{quickly} \\ \text{REST} & \boxed{3}\ \text{T} \end{bmatrix} \end{bmatrix} \\ \text{EOCRL} & \boxed{3} \end{bmatrix}$$

(5.31) **here**

$$\begin{bmatrix} \text{SUBCAT} & <\text{VP}_{\boxed{1}}> \\ \text{CONTENT} & \boxed{1}\ \text{RESTR}\ \boxed{3} \\ \text{EOCRL} & <\text{here}, \ldots> \end{bmatrix}$$

(5.32) **walks quickly here**

$$\left[\begin{array}{ll}
\text{CONTENT} & \boxed{1}\left[\begin{array}{ll}
\text{RELN} & \text{walk}\\
\text{WALKER} & \langle\ \text{SYNSEM LOCAL CATEGORY SUBCAT FIRST}\\
 & \text{SYNSEM LOCAL CONTENT INDEX}\ \rangle\\
\text{RESTR} & \boxed{2}\left[\begin{array}{ll}
\text{FIRST} & \text{quickly}\\
\text{REST} & \boxed{3}\left[\begin{array}{ll}
\text{FIRST} & \text{here}\\
\text{REST} & \boxed{4}\ \text{T}
\end{array}\right]
\end{array}\right]
\end{array}\right]\\
\text{EOCRL} & \boxed{4}
\end{array}\right]$$

The lexical entry of each adverbial specifies that its EOCRL is token-identical with that of the element it modifies. The instantiation of Schema 5 that licenses optional VP modifiers contains a stipulation that the mother's EOCRL is token-identical to the REST of the adjunct daughter's EOCRL. This allows the partially constructed restriction list to be passed up the tree without the EOCRLs being directly inherited between mother and daughter. In fact, it is crucial that the EOCRLs of a mother and daughter V node be allowed to be different, because they represent different points on the restriction list. For each, the EOCRL represents that portion of the restriction list which has not been instantiated for the V node at that level.

This approach has a number of desirable features. It allows the semantics of adverbs to unify with that of the element they modify, while requiring no modification to Schema 5, under which the mother in a head-adjunct construction has semantic CONTENT reentrant with that of the adjunct daughter. Multiple adverbials are allowed, even of a single adverbial type, while still providing the mechanism for ruling out multiple occurrences of adverbial types with non-contentful prepositions.

However, the mechanism for licensing the desired structures is not very elegant. The EOCRL feature has no purpose except to keep track of the end of the restriction list. It is clearly related to the CONTENT slot, yet it must sit outside of CONTENT, on the same level with, say, CONTEXT, with which it has no direct relation. In the next section, I propose a modification to Pollard and Sag's verb semantics which has the same advantages as the difference list approach without the clumsy mechanisms needed there.

5.3.2 A Modification to the Semantic CONTENT of Verbs

As noted in the first section, the reason that adverbs could not function the same way as adjectives to begin with was that the semantics of verbs is not like that of common nouns. In particular, CNs have a semantic content which always has slots INDEX and RESTR, while verbs have RELN and whatever other slots the lexical relation requires.

Having investigated what the semantics of adverbs would have to look like in order to produce the correct results with the given verb semantics, I would now like to shift the focus to how the semantics of verbs might be changed, in order to produce the correct results with adverbs whose semantics is more like that of adjectives.[6]

Very simply, I am proposing that the semantic CONTENT of verbs be an object with two slots, PSOA and RESTR, just like that of CNs. While the INDEX of a common noun is an index, the PSOA of a verb is a script, containing a RELN and other relevant slots, just as the CONTENT of a verb does in [Pollard and Sag, In press]. For example, the AVM of "walks" would be modified from that of Pollard and Sag's (11) on p. 20 (repeated as (5.33)) to (5.34).

(5.33) walks

$$\left[\begin{array}{ll} \texttt{CATEGORY} & \left[\begin{array}{ll} \texttt{HEAD} & \texttt{verb[fin]} \\ \texttt{SUBCAT} & \langle\, \texttt{NP[nom]}_{\boxed{1}[3rd,sing]} \,\rangle \end{array}\right] \\ \texttt{CONTENT} & \left[\begin{array}{ll} \texttt{RELN} & \texttt{walk} \\ \texttt{WALKER} & \boxed{1} \end{array}\right] \end{array}\right]$$

(5.34) walks

$$\left[\begin{array}{ll} \texttt{CATEGORY} & \left[\begin{array}{ll} \texttt{HEAD} & \texttt{verb[fin]} \\ \texttt{SUBCAT} & \langle\, \texttt{NP[nom]}_{\boxed{1}[3rd,sing]} \,\rangle \end{array}\right] \\ \texttt{CONTENT} & \left[\begin{array}{ll} \texttt{PSOA} & \left[\begin{array}{ll} \texttt{RELN} & \texttt{walk} \\ \texttt{WALKER} & \boxed{1} \end{array}\right] \\ \texttt{RESTR} & \texttt{end} \end{array}\right] \end{array}\right]$$

---

[6] Logically, there is another direction of investigation as well, namely changing the semantics of common nouns to be more like that of verbs. However, the functioning of adjectives has already been clearly spelled out in [Pollard and Sag, In press], while that of adverbs has not. It is not clear to me that there is any way to change the semantics of CNs to be more like verbs that would allow for a straightforward treatment of adjectives.

With this modification, adverbs can compose with their sister VPs in exactly the same way that adjectives do with nouns. The feature structure of "quickly," shown in (5.35), is now entirely parallel to that for adjectives.

(5.35) **quickly**

$$\left[\begin{array}{ll} \texttt{CATEGORY} & \left[\begin{array}{ll} \texttt{HEAD} & {}_{\texttt{adv}}\left[\begin{array}{ll}\texttt{MOD} & \texttt{V}:\left[\begin{array}{ll}\texttt{PSOA} & \boxed{1}\\ \texttt{RESTR} & \boxed{2}\end{array}\right]\\ \texttt{PRD} & -\end{array}\right]\\ \texttt{SUBCAT} & \langle\rangle\end{array}\right]\\ \texttt{CONTENT} & \left[\begin{array}{ll}\texttt{PSOA} & \boxed{1}\\ \texttt{RESTR} & \left\{\left[\begin{array}{ll}\texttt{RELN} & \texttt{quickly}\\ \texttt{ARG} & \boxed{1}\end{array}\right]\right\}\cup\boxed{2}\end{array}\right]\end{array}\right]$$

5.4 Conclusion

By investigating the semantics of adverbs in the theory of HPSG, I have motivated a modification to the current feature structure representation of the semantic content of verbs within the theory. Because restrictive modification applies not just to nouns and verbs, but also to adjectives and adverbs themselves, similar arguments can be constructed to motivate revisions in the feature structures of these categories as well. Here also, there must be separate nodes of the DAG for the parameter and the restrictions of the semantic content.

Such a revision to the feature structure of adjectives and adverbs, besides being empirically motivated, leads to a greater uniformity of representation among the various syntactic categories. As a preliminary generalization, we might hypothesize that all and only those categories which Pollard and Sag define as 'substantive' must have a structure consisting of RESTRICTION and something corresponding to INDEX or PSOA as their semantic content.

In addition, the revision proposed here may find independent motivation in the phenomenon of VP-pronominalization. It has been assumed in preliminary work on pronominalization in HPSG that referential pronouns must be of the type *referential-object,* to distinguish them from expletive pronouns. The difficulty then is to identify the part of the semantic content of the verb which is of the proper type to be referred to by a pronoun. The revision I have proposed in this section suggests a solution to this problem. The CONTENT of the verb is itself a referential object, with separate fields for parameterized state of affairs and restrictions. Further work is needed to determine whether VP-pronominalization can be captured as a structure-sharing of the PSOAs of a VP's semantic content.

Chapter 6

Hypothesis Construction

This chapter and the following one describe the algorithm and data structures for acquiring new lexical items from context. Chapter Six concerns the operation of the algorithm upon a single encounter with an unknown lexical item, while Chapter Seven describes the combination of information obtained from multiple encounters with the same new lexical item.

The first section of this chapter provides a brief overview of the hypothesis construction algorithm. The second focuses on how the system detects the need for lexical acquisition to take place, delimiting the type of lexical gaps covered in this dissertation. The third describes the acquisition of syntactic features of new words, and the fourth the acquisition of semantic features.

6.1 Overview

As described in Chapter Three, Gerdemann implemented several optimizations of Earley's algorithm in UNICORN [Gerdemann, 1991], one of which was to pass a set of predictions to the scanner. The scanner, responsible for identifying constraints on the feature structure of the next lexeme, uses this information to eliminate spurious lexical ambiguity whenever possible.

For example, encountering the lexeme "lives" in a context which predicts a noun but not a finite verb, such as (6.1), results in only the nominal sense being considered as a possibility. In a sentence such as (6.2), on the other hand, the context preceding the lexeme "lives" determines that only the verbal sense is retrieved. Without the use of restricted predictions, the scanner in both cases would have returned all of the possibilities for this item in the lexicon, including both the plural noun and third singular verb senses.

(6.1) We read about the lives of the presidents.

(6.2) One of the men lives in a furnished apartment.

There are, of course, contexts in which the predictions include both senses as possibilities. In the following sentences, at the point where the parser first encounters the word "lives," it could be either a plural noun or a third person singular verb.

(6.3) The other lives were lived well.

(6.4) The other lives in that house.

The learning procedure is integrated into the parser at the point of lexical lookup. When lexical lookup fails to identify a lexical entry appropriate to the current context, the learner constructs hypotheses based on the predictions. These hypothesized lexical entries are returned to the parser, and parsing continues. When parsing has been completed, the successful parses are examined to determine what additional information became instantiated on the hypothesized lexical items. In this way, the learner acquires feature structure information not represented in the restrictor or specified by the preceding context.

There are two stages in inferring the feature structure of an unknown word. First, the system constructs a set of hypotheses for its feature values, given the current context. This stage is described in the remainder of this chapter, and an outline of the algorithm is given in pseudo-code in Figure 6.1. These hypotheses are then compared with those which were constructed from previous contexts, and the hypothesis set is narrowed to those which fit both. The combination of hypothesis sets is described in the following chapter. This updated hypothesis set is then stored as the new set of hypotheses for the feature structure of the word.

6.2 Detecting an Opportunity for Learning

The first step in any type of machine learning is to detect the need for a learning episode. For an NLU system, this means a parsing failure. Therefore, a learning system must be able to recover gracefully from a failed parse, and work with the partial results. As described in Chapter Three, UNICORN uses a chart-parsing strategy. It builds a chart progressively during the parsing process, and if parsing fails, the partial chart is available for inspection. The predictions on the

```
Detect Opportunity for Learning from Parsing Failure
Establish Upper Bounds
    For each distinct set of predictions made at the point of failure by Earley's
    algorithm,
        Create a lexical entry consisting of the information in the restrictor.
        Add category type information to the lexical entry.
Combine Lower Bounds
    For each hypothesised upper bound,
        Continue parsing, to produce a set of instantiations of the upper
        bound,
        For each instantiation,
            If the instantiation is more informative than the upper bound,
            then make this instantiation an lower bound corresponding to the
            upper bound.
```

Figure 6.1: Algorithm for constructing a version forest from a single input.

chart at the point of failure can thus be used to constrain the feature structure of the unknown word.

Once a word is determined to be unknown, it is easy to identify the focus of the learning episode; the goal is to learn the unknown word. However, recognising that a word is unknown is not equivalent to simply determining that it does not exist in the lexicon. Following Earley's algorithm for parsing [Earley, 1970], Unicorn uses a combination of top-down and bottom-up strategies. Unlike a strictly bottom-up parser, it does not start with the input, look each word up in the lexicon, see what categories it belongs to, and use those categories in parsing. Instead, it predicts the possible categories at each step and scans for them.

This means that the parser does not recognise an unknown word merely by not finding it in the lexicon. Rather it looks for it in each of the predicted categories,[1] but no others, and determines that it is unknown if and only if it is not found in any of those categories.

---

[1] As explained in Chapter Three, the term 'category' here refers to all and only the categorial information specified in the restrictor. This most often includes the syntactic category, but generally a fair amount of other syntactic information as well, at the discretion of the grammar writer.

When a word is determined to be unknown, the learning mechanism again uses the predicted categories. But now, instead of looking up the word in those categories, the system hypothesizes that the word is a new lexical item of each of those categories, and determines for each whether it leads to successful parse results.[2]

6.3 Inferring Syntactic Category

Having identified the target of learning, the system must now determine the feature structure of the word to be learned. The general algorithm for inferring a feature structure involves constructing a type of version space of possible lexical entries for the new word. This algorithm is described in this section, while the following section deals with the instantiation of the semantic content of the new word.

6.3.1 Establishing Upper Bounds

There are two sources of information which are used to determine an upper bound, or starting point, for the feature structure of the new word: the predictions made by Earley's algorithm, and category type information supplied by the learning mechanism itself. Both types of information are used by the function `construct_lex_gram`, which builds a grammatical representation for the hypothesized lexical entry of the new word.

6.3.1.1 Using Information from the Restrictor

When an unknown word is encountered during parsing, sets of predictions have already been made by Earley's algorithm for certain of its features, namely those which have been specified in the restrictor. Each set of predicted features represents the minimum amount of information which must be present on a lexical item in order for it to satisfy the given context.

As mentioned in Chapter Three, the set of feature paths in the restrictor is determined by the grammar writer. Finding the most effective restrictor is to some degree a matter of trial and error. There is a balance between having too little information in the restrictor, which leads to insufficient constraints on the item being scanned in the lexicon, and too much information, which means that it takes longer to scan each item in the lexicon, without a corresponding pay-off of disambiguation.

---

[2] In a system whose goal is to find the single most likely definition of a new word, one possible way to increase the efficiency of the learning mechanism would be by ordering the hypothesized categories according to likelihood. For example, the system would start by guessing the highest level of category that could fill the prediction, then try expansion rules one by one. However, the current system is designed to find all possible definitions of a new word, so this optimisation has not been implemented.

The following restrictor is the one currently being used in lexical acquisition, and represents one acceptable compromise between these conflicting demands.

```
(6.5)   acqrestrictor :-
           change_restrictor([[maj],
                              [synsem, local, category, bar],
                              [synsem, local, category, subj],
                              [synsem, local, category, comps],
                              [synsem, local, category, head, maj],
                              [synsem, local, category, head, case],
                              [synsem, local, category, head, mod],
                              [synsem, local, content, restr, first, relation]]).
```

As these paths constitute the restrictor, they all contain predictions about the feature structure of any new word. However, some of these predictions may be vacuous. For example, the path SYNSEM|LOCAL|CATEGORY|HEAD|CASE is found only on nominal categories. Therefore, no value is predicted for it when the predicted category is something other than nominal. In such cases, the restrictor contains the special value "*?*," which is interpreted as a null prediction.

Other predicted categories are not eligible to be hypothesised as possible new lexical entries. For example, assume that the grammar includes the following PS rules.[3]

(6.6) $\mathrm{VP} \rightarrow \mathrm{V}_{[\mathrm{VFORM\ TRANS}]}\ \mathrm{NP}$

$\mathrm{NP} \rightarrow \mathrm{N}$

These rules predict that following a transitive verb, the next category may be either an NP or an N. However, it is a property of all lexical items that they must be lexical, by definition.

---

[3] This is a simplification, and somewhat of a misrepresentation, of the actual rules of the grammar, which are in HPSG notation. However, the form of the rules given here is easier to follow for this simple example.

Therefore, the lexicon does not allow lexical items of the category NP, so this prediction is ignored.

With the exception of predictions that are null or disallowed on lexical items, the paths specified by the restrictor are converted straightforwardly into path equations on the hypothesized lexical entry. As a result, there will be a separate hypothesized entry for each set of restrictions that is distinct in some significant way; *i.e.*, in some way other than lexicality or a path for which one of them makes no prediction.

6.3.1.2 Using Category Type Information

In addition to the information supplied by the restrictor, there is information which a lexical item must contain in order to be a valid lexical entry of a given syntactic category of English. This information is grammar-dependent, and must be stipulated by the grammar writer. Like the information from the restrictor, it is added in the component of the system which constructs hypothesized lexical items.

Some of this category type information is valid for all lexical items, regardless of their category. For example, the PHON feature of every lexical item has as its value a list, whose FIRST element is the surface form of the word itself, and whose REST is the end-of-list element `eol`. Similarly, all words have the path `<pecParseTree daughters word>`, whose value is also the surface form of the word. This path is not strictly part of the linguistic representation, but is an artifact of the system, used in displaying the tree structure. Feature-paths such as these, which are common to all lexical entries, are constructed for every hypothesized lexical entry.

Other information supplied by the lexical entry constructor is more category-specific. This is particularly the case with the paths that encode the semantic properties of the lexical item, which are distinct for each syntactic category, and yet constant for members of each category. For instance, when Earley's algorithm predicts that one of the possible categories of the new word is 'adjective,' the hypothesis constructor stipulates that the semantic content of the word consists of a set of restrictions, which includes the set of restrictions found on the modified noun. This, in accordance with [Pollard and Sag, In press], is part of what it means to be an adjective in English.[4] The addition of such features to guarantee well-formedness of learned lexical items amounts to an implicit encoding of type hierarchy information in the learning procedure.

The information predicted for a newly encountered lexical item by the restrictor, along with the typing information supplied by the hypothesis constructor, together constitute the minimal amount of information that must be contained in the lexical entry of this word. We refer to this set of information as the **upper bound** of information in the hypothesis space of a new word. For an unknown word appearing in a given context, there will be a set of upper bounds constructed for it, one upper bound for each distinct set of predictions made for the word in that context.

6.3.2 Establishing Lower Bounds

For each of the upper bounds, a lexical entry is constructed, containing all and only the information in that upper bound. The function that does this is named `construct_lex_gram`, because the form of a hypothesised upper bound is a very small grammar, consisting of a single lexical item.

Each of these lexical grammars in turn is compiled and asserted, so that the existing grammar is temporarily augmented by the information in the hypothesised lexical grammar. Parsing then continues, restarting from the point at which it failed previously due to the unknown word. If parsing still fails, then the hypothesised lexical entry must be incorrect, so the hypothesis is discarded. If all of the hypotheses lead to parsing failure, then the system concludes that the sentence is ungrammatical for reasons that have nothing to do with the unknown word.[5]

If parsing succeeds with a given hypothesised lexical item, then the system concludes that it is at least possible that that hypothesis may be correct. The system may be able to deduce quite a bit more than that, however. In the process of parsing by means of a unification grammar, features and values not specified in the original upper bound may well be assigned values. If a given upper bound leads to more than one successful parse, then different features may be assigned values, or the same values may be assigned in different ways.

---

[4] In reality, this generalisation only holds for a certain class of adjectives, those which are attributive and restrictive. A completely different set of restrictions would have to be stipulated for adjectives such as "alleged," "former," and "fake." I do not deal with these issues in this dissertation.

[5] This can only occur when there is something ungrammatical about the sentence at some point following the unknown word. If the anomaly occurred before the unknown, the parser would have failed before reaching the unknown.

For each of these features which is assigned a value in a successful parse, that feature-value may or may not be an essential part of the correct definition of the new word. For example, suppose that the system is acquiring new determiners, and does not know the words “a” or “the.”[6] On encountering these words in contexts such as (6.7) and (6.8), the system constructs upper bounds for the hypothesised lexical entries of these words. Given a reasonable grammar of English, these hypotheses would include the syntactic categories of noun and adjective, as well as determiner, though only the latter is relevant for this example.

(6.7) Kim saw a boy.
(6.8) Kim saw the boy.

For both words, the upper bounds would include those paths common to all lexical items and those common to all determiners, as well as any supplied by the restrictor. Parsing continues, and in both cases, successful parses are found, with “a” and “the” as determiners.[7] In both cases, the lower bounds resulting from the successful parses include the information that these determiners select for a following nominal constituent which has, among other things, the feature-value [NUMBER SINGULAR].

For the word “[illegible] we know, as speakers of English, that this is a part of the definition of the word, while for the word “the” it is not. This is evidenced by the contrast between (6.9) and (6.10).

(6.9) * Kim saw a boys.
(6.10) Kim saw the boys.

However, until the system has encountered examples such as (6.10), and failed to encounter examples such as (6.9),[8] it does not have any way to distinguish between the words “a” and “the.” As is shown in the following chapter, it is exactly such multiple encounters with a single unknown word that are necessary in order to determine the set of feature paths that are part of the definition of a word. Upon a single encounter with a word, it is often not possible to tell which of the feature values instantiated in a successful parse are properly part of the definition of the word, and which are artifacts of the particular context in which it is first seen.

---

[6] In practice, determiners are one of the least likely categories for which unknown lexical items will be encountered. However, this example is a simple case of a situation that occurs with other categories as well.

[7] The other hypotheses, such as adjective and noun, will also be tested, but will not lead to any successful parses in these examples.

[8] As discussed in Chapter Two, the system receives no negative data, so it will not be presented with the fact that sentences such as (6.9) are ungrammatical. This is discussed further in Chapter Seven.

Therefore, the instantiation of the word in a successful parse represents the lower bound of the feature structure of the new word. So to each upper bound, there corresponds a set of one or more lower bounds, since if there are zero lower bounds, no successful parses, the corresponding upper bound is rejected. The actual target definition of the word is known to be intermediate in specificity between an upper bound and one of its corresponding lower bounds. However, it is not yet known which lower bound, if there is more than one, or exactly how much of the information of the lower bound the target definition contains.

The hypothesis set resulting from successful parses with a given upper bound can be represented in a data structure consisting of an upper bound and a set of lower bounds. This data structure resembles a **version space**, discussed by [Mitchell, 1977]. However, a version space may have any number of upper bounds, while this structure has only one. In order to make this distinction, we refer to this data structure as a **lexical version space**. The lexical version space for the unique successful upper bound of the word "a" in the previous example is given in Figure 6.2.

In this simple example, there is just a single upper bound, and a single lower bound, which differs from the upper bound by a single additionally specified path. A slightly more complex example is necessary to show the full use of lexical version spaces as a data structure. One such an example is (6.11), in which "glarf" is assumed to be an unknown word.

(6.11) John glarf left.

For this example, let us assume the following simplified grammar fragment.[9]

```
(6.12)  S  → NP VP
        VP → V VP
        VP → V ADV-OBJ
```

---

[9] I am here making no attempt to express the example grammar in HPSG format, but simply use familiar PS rules, with the addition of simple feature information, for expository purposes. The abbreviatory conventions here should all be obvious, except for ADV-OBJ and MOD-ADV, which stand for "adverbial object" and "modifying adverb" respectively.

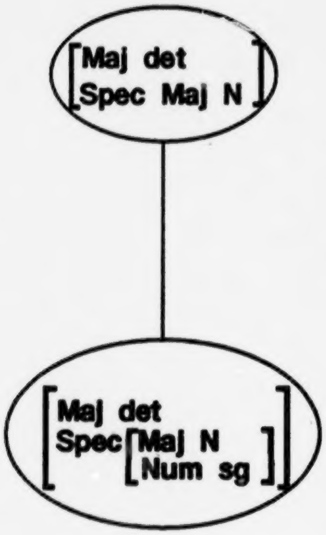

Figure 6.2: Lexical version space for a newly encountered determiner.

```
VP → MOD-ADV VP
VP → INTRANS
NP → NP CONJ NP
NP → John
INTRANS[PAST] → left
INTRANS[PPART] → left
ADV-OBJ → left
```

At the point at which the parser encounters the word "glarf," Earley's algorithm makes a number of predictions about what the syntactic category the next word of the input be a member of. First, "glarf" may be a conjunction, so that "John is the first element of a coordinate structure. Therefore, an upper bound is created for this possibility, and parsing continues with the grammar augmented to include the hypothesized lexical entry. However, no successful parses result from this augmented grammar, so the first hypothesis is retracted, and other possibilities are explored.

A second possibility is that "glarf" may be an adverb, syntactically similar to "quickly." Since there is an entry for "left" as an intransitive verb, this possibility leads to a correct parse. Since there is exactly one successful parse, there is a single lower bound corresponding to this upper bound, with all the information instantiated on the lexical entry as a result of unification in the parsing process.

Finally, "glarf" may be either an auxiliary verb or a verb which selects for an adverbial object. These possibilities correspond roughly to "John had left" and "John turned left" respectively. Whether these are treated as a single hypothesis or as two distinct hypotheses depends on the grammar and the restrictor. In both the sample grammar given here and the HPSG grammar used with the learning mechanism, auxiliaries are treated as verbs which select for VP sisters.[10]

The restrictor used with the learning mechanism, repeated in (6.13), does not discriminate according to the syntactic category of items on the subcategorization list, given in the path SYNSEM|LOCAL|CATEGORY|COMPS, but only according to whether the path has a non-null value. In other words, the restrictor does not distinguish verbs according to what kind of objects they take, but only according to whether they take objects or not.

(6.13)
```
acqrestrictor :-
    change_restrictor([[maj],
                       [synsem,local,category,bar],
                       [synsem,local,category,subj],
                       [synsem,local,category,comps],
                       [synsem,local,category,head,maj],
                       [synsem,local,category,head,case],
                       [synsem,local,category,head,mod],
                       [synsem,local,content,restr,first,relation]]).
```

---

[10] The reader may observe at this point that morphological clues could be used to give some indications of the syntactic category of the unknown. While we may investigate this possibility in future work, we have chosen not to do so at present, because such clues represent tendencies, rather than absolutes. While a form ending in "-ly" is frequently an adverb, and one ending in "-ed" frequently a past tense verb, there is not a one-to-one correspondence, due to pervasive irregular morphology in English. The tendencies that morphological clues show would be useful in a system whose task was to find the single most likely category of an unknown word. However, since the present system investigates all possibilities predicted by the grammar rules, such heuristics are not used.

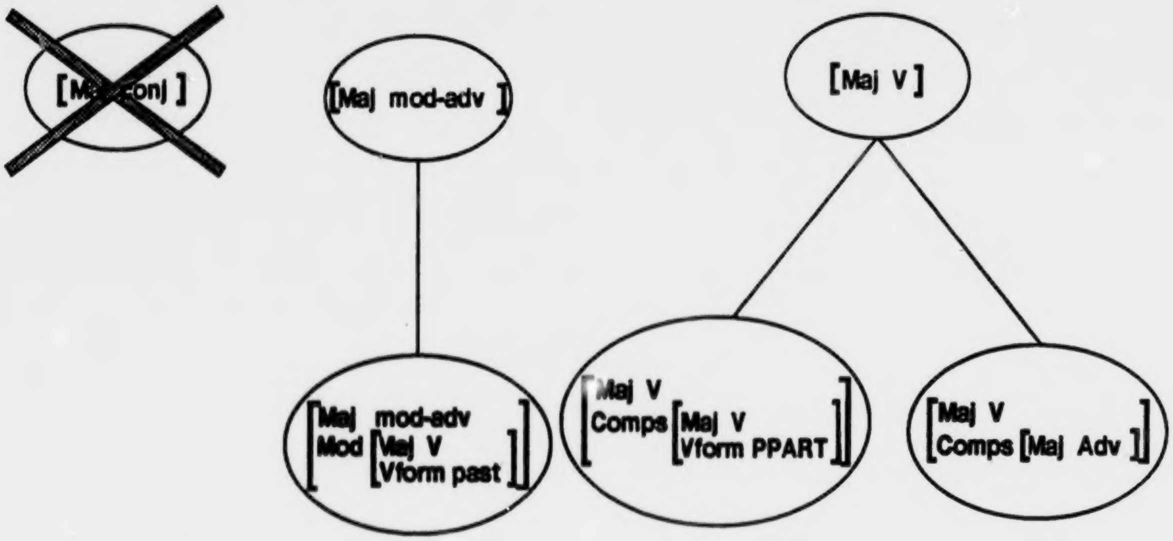
[Maj conj]
[Maj mod-adv]
[Maj V]
[Maj mod-adv Mod [Maj V Vform past]]
[Maj V Comps [Maj V Vform PPART]]
[Maj V Comps [Maj Adv]]

Figure 6.3: Abbreviated lexical version multi-space for the unknown "glarf" in "John glarf left."

Therefore, auxiliaries and other types of verbs, including those with adverbial objects, are currently treated as a single prediction set. There is therefore one upper bound constructed to hypothesise that "glarf" may be a verb. However, this hypothesis produces two distinct successful parses of the data, one in which "left" is an intransitive verb with the feature [PPART] (for 'past participle'), and the other in which "left" is an adverbial object. These parses instantiate feature paths on "glarf" in different ways. Therefore there are two distinct lower bounds corresponding to this one upper bound.

The resulting data structure is shown in Figure 6.3. Three distinct upper bounds are constructed, but one is discarded when it fails to lead to a successful parse. Of the other two, one has a single corresponding lower bound, while the other has two distinct corresponding lower bounds.

As with Mitchell's version spaces, the lower and upper bounds here determine a full space of possible concepts, with other possibilities intermediate in the amount of information they contain. A full diagram of the possibilities abbreviated in Figure 6.3 is shown in Figure 6.4. This is fairly simple example, and so the diagram is still fairly easy to follow. However, for more complex example, a full representation of the information abbreviated by the lower and upper bounds would be prohibitively complex.

In the work on concept learning with version spaces, the target concept is known to reside somewhere within a single version space. While our lexical version spaces are simpler than version spaces in having only a single upper bound, the situation is complicated by the fact that the target lexical entry resides somewhere in a disjunction of lexical version spaces, which we refer to as a **lexical version multi-space.** The challenge in dealing with multiple encounters with the unknown word is to combine these lexical version multi-spaces in a meaningful way. This is the topic of the next chapter.

The use of lexical version spaces described here enables the system to capture an important fact about the occurrences of new words. Namely, distinct tokens of the same surface form of a word may in fact be instances of the same lexical entry, or of different lexical entries, although they will almost invariably be instantiated with different feature paths in different contexts. That is, each token of a surface form will be represented by a lexical version multi-space of hypotheses about its definition. If these lexical version multi-spaces contain spaces representing the same syntactic category, then the multi-spaces of two tokens contain spaces which share compatible upper bounds, but generally incompatible lower bounds. Whether these two tokens represent the same or distinct lexical entries depends on whether or not the actual definition of the word(s) lies in the portion of the lexical version space which is compatible for the two.

This approach is particularly compatible with the framework of HPSG. In [Pollard and Sag, In press], it is claimed that there is no such thing as an optional complement, which entails that a verb with more than one possible subcategorisation frame will have more than one lexical entry. This is in the spirit of Carter's Core Language Engine [Carter, 1989], where verbs with different subcategorization frames are represented as unrelated lexical items.[11]

Lexical version multi-spaces thus have the advantages of Webster and Marcus's idea of Structured Overcommitment [Webster and Marcus, 1989]. Their system builds a lexical entry based on the first context where a new word occurs. Any arguments the word takes in this context are assumed to be obligatory, any that it doesn't have are assumed to be forbidden, until

---

[11] Thus optional elements are necessarily adjuncts rather than complements. Adjuncts are not subcategorised for, but rather themselves select the elements which they modify.

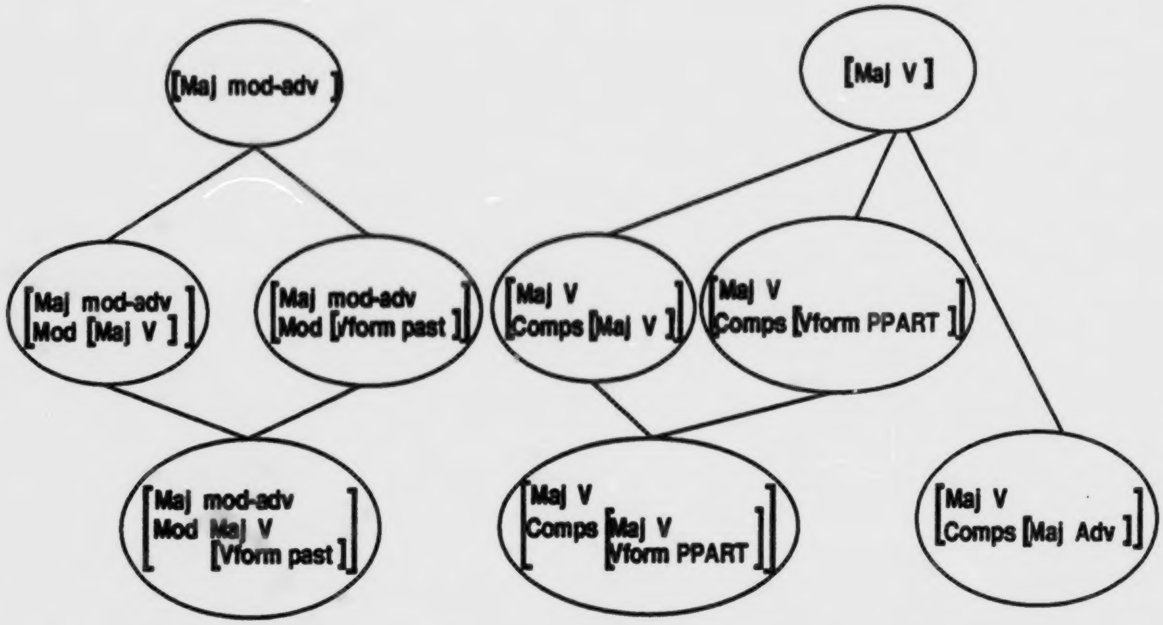
[Maj mod-adv]
[Maj V]
[Maj mod-adv Mod [Maj V]]
[Maj mod-adv Mod [Vform past]]
[Maj V Comps [Maj V]]
[Maj V Comps [Vform PPART]]
[Maj mod-adv Mod [Maj V Vform past]]
[Maj V Comps [Maj V Vform PPART]]
[Maj V Comps [Maj Adv]]

Figure 6.4: Full lexical version multi-space for the unknown "glarf" in "John glarf left."

another example indicates otherwise, at which point the lexical entry must be reconstructed. The use of lexical version multi-spaces maintains the overcommitted entry as one possibility in a hypothesis space, but does not regard it as necessarily correct until warranted by the evidence.

6.4 Inferring Semantics

In this section, I discuss how the system infers the semantics of the new lexical item. This is done via a combination of linguistic context and real-world knowledge. I first discuss the type of semantic knowledge which is the focus of this dissertation. Then I describe how the algorithm uses the real-world knowledge base to instantiate this type of information. This is followed by a brief discussion of extensionality vs. intensionality in the acquisition of semantic knowledge.

6.4.1 Description of Semantic Knowledge

The terms 'semantics' and 'meaning' have been used in a number of different ways by different researchers, to denote both linguistic and real-world knowledge. In this work, I am focusing on one particular aspect of semantics, the establishment of a link between a linguistic entity and an existing mental concept. In doing so, I am making certain claims and assumptions, which are set forth in this section.

First, I am assuming that there are certain aspects of semantics which are encoded in the feature structure of linguistic elements. In the most recent version of HPSG, [Pollard and Sag, In press], this is reflected in the fact that syntax and semantics are not treated as separate portions of the feature structure, but under a single path, SYNSEM. The type of semantic knowledge encoded in this way includes the relationship between the semantic content associated with two daughters nodes and the semantic content associated with the mother node.

For example, there is a portion of the feature structure of an attributive restrictive adjective, in particular its SYNSEM|LOCAL|CONTENT value, which specifies that its semantic content, and hence the semantic content of the node immediately dominating it in a *head-modifier* structure, contains an index, token-identical with that of the nominal it modifies, and a set of restrictions, which is the union of a lexically specified restriction with the set of restrictions on the modified nominal.

Second, I am assuming that there are aspects of what is often called 'meaning' that do not have any linguistic representation whatsoever, and thus do not need to be learned by a lexical acquisition system. This does not by any means imply that these aspects are irrelevant for semantic interpretation or disambiguation. On the contrary, they often play a crucial role in such matters for human language users. However, what this demonstrates is the interaction of linguistic and non-linguistic knowledge, rather than the need to encode all knowledge in the lexicon.

Knowledge of this type is often called encyclopedic knowledge, and stands in roughly the same relationship to an encyclopedia as lexical knowledge does to a dictionary. One often-cited example of such knowledge is knowledge about simple arithmetic. This is what makes sentences such as (6.14) acceptable, while sentences such as (6.15) are somehow jarring.

(6.14) Five thieves conspired to murder a sixth.

(6.15) ? Five thieves conspired to murder a twelfth.

I am taking the position that (6.15) is perfectly good not only syntactically, but from a formal semantic viewpoint as well. Whatever may be wrong with it has to do with people's understanding of the number system, and should not be addressed as a linguistic phenomenon.

Another example of encyclopedic knowledge is found in the following sentence, which has an ambiguity that may go unnoticed by many native speakers.

(6.16) Friends resist the urge to say bitter words for the sake of preserving the friendship.

Syntactically, the phrase "for the sake of preserving the friendship" can modify either "say bitter words" or "resist the urge to say bitter words." Deciding that it actually modifies the latter requires, as a rough sketch, at least the following information.

- Bitter words are likely to be unpleasant to the hearer.
- The bitter words referred to here represent an utterance in which the speaker and the hearer are the friends denoted by the subject and their partners in friendship, respectively.
- The friendship referred to here is that between the friends denoted by the subject and the hearers of words which they utter in some counterfactual world.

- Resisting an urge to do something entails not doing it.

- A goal of preserving a friendship is generally better served by not doing something unpleasant to the partner in friendship than by doing it.

At least some of these pieces of information, especially the last, represent knowledge about the world which is not associated with any particular lexical item. Therefore, it is not the kind of information which should be acquired by a lexical acquisition system. It is, however, crucial to the interpretation of the sentence. The very fact that people deal with such information so easily is what makes such ambiguities often slip by unnoticed. In the current system, I do not attempt to resolve such ambiguities, as the knowledge required to do so is outside the scope of the system.[12]

However, there is another kind of information which is also often referred to as real-world knowledge, which is encoded in the knowledge base of the system. As described in Chapter Three, this is semantic type information. This includes information about what kinds of entities in the real world may fill the roles of what scripts, or what kinds of properties may be predicated of what types of entities. This information is also outside of the lexicon itself, but is used for acquisition, as well as for interpretation and disambiguation.

Thirdly, for the purposes of this work, I am assuming that the acquisition of knowledge about the world is at least partially independent from the acquisition of knowledge about language. That is, it is possible for people, as well as for computer systems, to acquire a new concept without knowing the word for it, or to acquire a great deal of information about a word without knowing anything about the concept it denotes. In order to be able to communicate effectively about mental concepts, both types of knowledge must be present, and the awareness of one type of knowledge without the corresponding knowledge of the other type may provide motivation to fill in the gaps in knowledge, either of linguistic entities or of mental concepts. However, knowledge of one type may often precede the other.

For example, a person may encounter a new kind of flower, a previously unknown food, or an emotion that they do not have the words to describe. New mental concepts are created to represent these entities, based on encounters with the real world or on mental manipulation of existing concepts. It is natural for a person to seek a linguistic tag for such entities, asking questions such as "What's this thing called?" But the answer to such questions is not prerequisite to the formation of a mental concept for the entity.

---

[12] The encoding of such encyclopedic knowledge is the goal of the Cyc system, as described in [Lenat and Guha, 1990]. This system is currently being used for very powerful inference generation of a number of kinds, but it is not clear to what extent this has been integrated with issues of Natural Language Understanding.

Conversely, a person, or computer system, may encounter a new word, and be able to acquire quite a bit of information about it, without having a mental concept to match it up with. For example, in the following utterance, the hearers may not know the real-world concept to be associated with the word "cogitate."[13] However, they will be able to conclude that it is an intransitive verb, and that it represents some type of activity.

(6.17) It's when you're not doing much of anything, just sort of sitting around cogitating, wondering what things might be like someday.

In this example, at least for the context in which it occurred, it could be argued that all of the hearers already had the mental concept denoted by the word "cogitate."[14] What they lacked was the knowledge that this particular word is linked to their existing concept.

The study of how people or machines acquire new concepts about the real world is outside the scope of this work. Therefore, I am making the assumption that the concepts to be linked with new words all already exist in the knowledge base. What the learning system acquires, in addition to syntactic information, is the type of information that the hearers of (6.17) lacked: the link between a newly encountered word and a known concept of something in the real world. How this is done is described in the next subsection.

6.4.2 The Method of Acquiring Semantic Knowledge

As described previously, for each syntactic category, the grammar specifies the semantic structure of that category. Once the syntactic category of a hypothesized lexical entry is determined, from the restrictions of Earley's algorithm, the paths representing the semantic portion of the feature structure are known, and can be added to the hypothesized lexical entry. This is done in the current implementation by the function `add_content_paths`, which takes a syntactic category as input, and writes the paths for the semantic structure associated with that category onto the hypothesized lexical entry.

---

[13] This utterance was produced during a game of 'Taboo.' The goal of the speaker was to get the hearers to guess the word "daydream," without uttering any of a number of other words, such as "think," "ponder," etc.

[14] This is not completely uncontroversial. It is still an open question whether any two words, such as "cogitate" and, say, "meditate," can be exact synonyms, and if not, what different relationships they may have with mental concepts. I consider this to be a problem of philosophy of language, and will not address it further here.

For each syntactic category, and thus for each type of semantic structure, there is a portion of the structure which is specified lexically, that is, not inherited from any higher node of the lexical hierarchy. This represents the unique contribution of the lexical item itself, that part of its meaning which is independent of the meaning of the elements it combines with.[15]

As an example, the clause of the function `add_content_paths` which supplies the semantic content paths of an unknown attributive restrictive adjective is given in (6.18).

```
(6.18)    add_content_paths(adj,Stream) :-
              nl(Stream),
              write(Stream, '< synsem local content index > =
                             < synsem local category head mod synsem local
                             content index >'),
              nl(Stream),
              write(Stream, '< synsem local content restr first > =
                             < synsem local category head mod synsem local
                             content restr first >'),
              nl(Stream),
              write(Stream, '< synsem local content restr rest first arg >=
                             < synsem local category head mod synsem local
                             content index >'),
              nl(Stream),
              write(Stream, '< synsem local content restr rest first relation >
                              = Glarf'),
              nl(Stream),
              write(Stream, '< synsem local content restr rest rest > =
                             < synsem local category head mod synsem local
                             content restr rest >').
```

---

[15] It is possible that a few syncategorematic elements have no such contribution. For example, the "to" which marks infinitives apparently makes no semantic contribution, but has only a syntactic function, marking the element it combines with as having the feature-value [VFORM BASE] and the mother which immediately dominates them both as [VFORM INFINITIVE].

Note that with one exception, all of the paths in this structure are fully specified, with values which represent structure-sharing with other paths. For example, the first path equation here says that the index in the semantic content of this word is structure-shared with the index in the semantic content of the element which this word modifies. Again, within the theory of HPSG this part of what it means to be an attributive restrictive adjective. All of the structure-sharing paths in this function are of this type.

The exception is the path SYNSEM|LOCAL|CONTENT|RESTR|REST|FIRST|RELATION. This specifies the relation on the second element of the set of restrictions which constitutes the semantic content of nominal constituents. Its value, here given as "Glarf," is an arbitrary variable; for an implementation in prolog, it could be any string beginning with a capital letter.

This path represents the lexical contribution of this particular word. It is the one path whose value is different for every lexical item of this syntactical category. The fact that all other paths are instantiated indicates that for the given hypothesis, the syntactic category has been determined, but not the particular mental concept which this word is linked to.

The next step of the algorithm is to instantiate this variable, in as specific a way as possible. From a single context, it is often not possible to determine a unique concept with which a given word is linked. Therefore, the algorithm determines the full set of possible concepts for the word. The next chapter describes how information from multiple encounters with the same word is combined to narrow down the range of possible semantic links for a given word.

As described in the previous section, when a hypothesised lexical entry has been constructed for the new word, parsing continues, with the grammar being augmented by the new entry. This entry contains the semantic paths determined for the new word, including a path whose value is a variable.

The parsing process interleaves semantic checking with the proposing of syntactic rules as candidates for parsing. Semantic checking, described in Chapter Three, involves verifying that the daughters of a proposed PS rules denote entities whose types can combine felicitously, according to the relationship specified for them by the PS rule.

For example, suppose that the system is parsing the input in (6.19), with the prediction that an N2 is to come next. Abbreviated forms of the PS rules used in this example are those given in (6.20). However, the full forms of these rules in the HPSG grammar contain much more information about the semantic relationships among the elements, which are relevant to the example.

(6.19) a glarf driver

(6.20) `N2 → Det N1`
`N1 → Adj N1`
`N1 → N`
`Det → a`
`N → driver`

After predicting a determiner and successfully scanning for one, the parser will next predict an adjective, and construct a hypothesised lexical entry for this prediction.[16] It then predicts an N, and successfully scans for one, finding the noun "driver."

However, before the rule `N1 → Adj N1` can be successfully completed, the semantic checker is consulted, to determine whether the entities denoted by the daughters can properly stand in the relationship specified by the rule. In this case, that relationship is the the `Adj` daughter denotes a property which is predicated of the entity denoted by the `N1` daughter.

Therefore, in order for this rule to be accepted as a candidate in the parse of this phrase, "glarf" must denote a property which can be predicated of a driver. The set of such properties is the set of ways in which the variable in the lexical entry of "glarf" can be instantiated in the prolog clauses used in semantic checking.

The closed world assumption inherent in a limited domain is crucial for such a system. There is a finite set of properties which can be predicated of drivers in the small domain modeled in the knowledge base, and only these are proposed as instantiations for the meaning of "glarf" in this example.

In fact, there are no properties which can be predicated only of drivers, according to the current knowledge base. However, a driver is known to be a type of person, and there are a number of properties which can be predicated of persons. Thus the SYNSEM|LOCAL|CONTENT|RESTR|REST|FIRST|RELATION path of "glarf" in this example would be instantiated as the following set.

(6.21) [belligerent,dead,disinterested,eastbound,northbound,southbound,unable,westbound]

---

[16] It will also predict a noun, and construct a hypothesised lexical entry for "glarf" as a noun. This possibility, which ultimately fails, is not discussed in detail here.

As another example, on first encountering the word "glorp" in a sentence such as (6.22), and having inferred that it is a transitive verb (as one possibility), the system should determine that it denotes an action involving a driver and a street. That is, it denotes an action which can take as one of its roles the entity denoted by the phrase "the street" and as another the entity denoted by "the driver."

(6.22) The officer observed the driver glorp the street.

The information from this sentence restricts the set of possible denotations of the word considerably, since there is a relatively small set of actions, compared to the universe of possible actions, that a driver can perform with a street. After a first encounter, the system knows that the action denoted by "glorp" is one of this small set, though it does not know which precise action it is. Further input is used to narrow the range of possible meanings, as is described in the following chapter.

6.4.3 Extensionality vs. Intensionality

The current system for inferring semantics is extensional rather than intensional. That is, it instantiates the variable in the semantic feature structure of the unknown as a disjunctive set of atomic values. An intensional system would be one which characterized the set of possible values for the variable, without necessarily listing them all, such as "properties which may be predicated of a person."

There are advantages and disadvantages to both intensional and extensional systems, and each can capture important facts that the other misses. An extensional system is generally faster and more efficient in situations where a relatively small number of possible atomic values are involved, as is the case with the current knowledge base.

The goal of the lexical acquisition system is to determine the possible meanings of a new word. When such a goal is not uniquely achievable on a single encounter with the new word, the extensional system lists the possible solutions licensed by the given data, while an intensional system would give a description of all such solutions. As new constraints are inferred from further data, the convergence toward a unique solution will be clearer in the extensional system, which will produce successively smaller disjunctive sets of possible solutions. An intensional system would produce successively more restrictive lists of constraints, which would similarly restrict the possible solution set, but in a way which is less obvious to the user. In fact, complete convergence on a solution, represented by a singleton set of solutions in the extensional system, is not signaled in any obvious way with an intensional system.

For these reasons, I have chosen to make the semantic acquisition operate extensionally. However, this may not be feasible with a system operating outside of a restricted domain. In a system which contains information about a very large number of persons, physical objects, or properties, disjunctive lists of such things become unwieldy. Moreover, the intensional system, by its nature, captures generalizations which are missed by the extensional.

An example of the differences between the extensional and intensional approaches is shown in the following sentences, which, taken as successive pieces of input, lead to the inference of two constraints on the meaning of "glump."

(6.23) The driver continued to go straight for three glumps.

(6.24) The driver went around the glump three more times.

The word "for" in (6.23) indicates that, as one possibility, "glump" is some type of measure.[17] The extensional system determines that "glump" must denote one of the finite set of measures in the knowledge base, given in (6.25), while an intensional system would characterize this set as denoting a measure.

(6.25) [block,foot,mile]

In (6.24), "glump" is the object of the preposition "around," which, at least on one reading, selects for a complement that denotes a physical object. The extensional system returns a disjunctive set of all physical objects in the knowledge base, which is given in (6.26). The intensional system, on the other hand, would again return a characterization of this set.

(6.26) [block,cadillac,car,corner,curb,fender,grinder,pole,rear,sheet,side,stopsign,tire,traffic, truck,vehicle]

In combining these two constraints, the extensional system simply takes the intersection of the disjunctive set.[18] For this example, this produces the singleton set [block], which is now determined to be the unique possibility for the meaning of "glump." The intensional system, on the other hand, produces a list of constraints, all of which the referent of "glump" must satisfy. This expresses the generalizations that the referent of "glump" must be something which is both a physical object and can be used as a measure. However, it fails to capture the fact that, in the given knowledge base, there is a unique entity which satisfies both of these constraints.

---

[17] Other possibilities arise from the ambiguity of "for," which can have not only an extent interpretation, but can mark its object as a recipient, benefactive, substitution, or any of several others. Similar ambiguities arise with "around" in (6.24). As such ambiguity is a general property of the grammar and lexicon, I will not discuss all of these readings here, but simply note that the system will consider all of them to be possibilities.

[18] Further details on this process and discussion of special cases are given in Chapter Seven.

Again, an extensional system is not feasible for a very large knowledge base. Therefore, even if the current learner uses an extensional approach, it should be easily convertible to an intensional system. This is the case with the current system. In fact, the prolog clauses that form the semantic hierarchy in the knowledge base reflect exactly the information that an intensional system works with. The only thing that makes the current system extensional is that once the variables are instantiated by atomic values which satisfy all the given constraints, those constraints are no longer visible to the user.

All that would be required to make the current system intensional, then, would be to maintain a record of the nodes of the semantic hierarchy that are consulted in the successful instantiation of the semantic variables. This will be a topic for future work, as the knowledge base grows beyond the point where an extensional system is practical.

Chapter 7

Hypothesis Combination

On any single encounter with a new word in context, it is likely that there will not be sufficient information to assign it a unique feature structure. However, multiple encounters with the same unknown in different syntactic environments provide a larger number of constraints which the unknown must satisfy. Therefore, combining the constraints deduced from multiple inputs enables the system to (partially or completely) converge on a syntactic representation for the new word.

This convergence happens in several ways. On each encounter with a given unknown word, a lexical version multi-space is created, representing the possibilities for the feature structure of the word in the current context. This lexical version multi-space is combined with that which resulted from earlier encounters with the word, and the combination becomes the updated lexical version multi-space representing the hypothesis space of definitions of the word. As a result of this combination, the number of upper bounds may reduced, so that the updated lexical version multi-space contains fewer lexical version spaces than either the lexical version multi-space of the current context or that resulting from previous encounters with the word. Within a lexical version space which is shared by the current and previous lexical version multi-spaces, the upper bounds may be further instantiated, and the lower bounds generalized or eliminated, thus pruning the hypothesis space of possible definitions.

This chapter describes how multiple lexical version multi-spaces are combined to produce an updated lexical version multi-space. As an overview, an outline of the algorithm is given in Figure 7.1. The first section describes the combination of upper bounds, and the second the combination of lower bounds. The third section demonstrates the behavior of this algorithm with some simple examples, while the fourth section discusses the issue of convergence on a completely specified definition of the new word.

7.1 Combining Upper Bounds

The lexical version multi-space constructed during an encounter with an unknown word, and the lexical version multi-space representing the previous hypotheses about the word each contain a set of lexical version spaces, where each lexical version space has exactly one upper bound.[1] Each upper bound represents a prediction made about the unknown.

The first step in combining the previous and current lexical version multi-spaces is to combine their respective sets of upper bounds. This combination forms the set of upper bounds for the updated lexical version multi-space. The procedure for combining the two sets of upper bounds is described in this section.

7.1.1 Cross-Unification of Upper Bounds

Intuitively, the set of upper bounds in the updated, or output, lexical version multi-space should be something like the intersection of the sets of upper bounds in the two input lexical version multi-spaces. However, instead of saying that an element is in the output set iff it is **token-identical** to elements in each of the two input sets, we want to say that an element is in the output set iff it represents the successful **unification** of a pair of elements, one from each of the two input sets. To construct the output set, we try to unify each element of one set with every element of the other set. The successful (non-null) unifications make up the members of the output set. We call this operation **cross-unification**, from its similarity to taking a cross-product.

According to this definition, in the general case, the output set of upper bounds could be larger than the input sets. Since each element of one set is compared with every element of the other set, if all the attempted unifications were successful, the size of the output set would be the product of the sizes of the input sets. However, with reasonable assumptions about the feature structure information found in the sets of upper bounds, each element of one set

---

[1] On the first encounter with the word, the previous lexical version multi-space is the null set.

```
Combine Upper Bounds
  For each element in the set of upper bounds of the new lexical version
   multi-space,
    For each element in the set of upper bounds of the previous lexical
    version multi-space,
      If the two upper bounds have a non-null unification
        then put their unification into the set of upper bounds for the
        updated lexical version multi-space.
Combine Lower Bounds
  For each pair of upper bounds whose unification was added to the set
  of upper bounds of the updated lexical version multi-space,
    Retrieve the corresponding two sets of lower bounds.
    For each element in the new set of lower bounds,
      For each element in the previous set of lower bounds,
        Determine the information shared by the two lower bounds.
        If the shared information is equivalent to the upper bound,
          or the shared information is equivalent to an existing lower
            bound,
          or the shared information subsumes an existing lower bound,
        then discard it.
        else add it to the set of updated lower bounds.
```

Figure 7.1: Algorithm for combining lexical version multi-spaces.

generally unifies with at most one element of the other set. This is because the elements of both sets are created from predictions made using the same restrictor. They will therefore be specified for largely overlapping information, such as syntactic category.

In order for two upper bounds to have been established as distinct within a lexical version multi-space, it must be the case that either they are specified for different paths, which is relatively rare, given that they were constructed with the same restrictor, or they receive different specifications for at least one of the same paths. Therefore, if a member of one set of upper bounds unifies with a certain member of the other set, it will most likely not unify with any other member of that other set. This is why cross-unification often behaves like intersection in this domain. As a result, the updated set of upper bounds is generally either the same size as or smaller than the smaller of the two input sets of upper bounds.

Because the members of the two input sets are created with the same restrictors, but in different contexts, they most often have feature structures which are either identical or non-unifiable. The case where two upper bounds are unifiable but not identical occurs when a path for which the restrictor predicts a value in one context has no value predicted for it by the same predictor in a different context. For example, this will be the case with path SYNSEM|LOCAL|CONTENT|INDEX|NUMBER of the unknown word "glarf" in the following contexts, assuming that all other words in the sentences are known.

(7.1) The driver saw a glarf.

(7.2) The driver saw the glarf.

In (7.1), the preceding context, specifically the determiner "a," predicts that "glarf" must be singular. The context of (7.2), however, makes no prediction about the number of "glarf." Otherwise, the predictions made in these two cases are identical. Therefore, the upper bounds for the lexical version spaces of the hypotheses constructed for the prediction that "glarf" is a noun in these two examples are unifiable, but not identical.

What the system is trying to determine from these examples is a single definition of the word "glarf" which satisfies all the constraints of all the contexts in which it has been seen. Therefore, the updated upper bound must reflect all of the information in both of the input upper bounds, so long as their specifications are consistent. The complete set of compatible constraints in the two input upper bounds is represented by their unification.

In this example, the unification of the two input upper bounds is identical to the upper bound constructed for (7.1), the more fully instantiated of the two. In the more general case, it may happen that two upper bounds are unifiable, and their unification is not identical to either, because each contains some path not found in the other. However, I have not encountered such a case in actual input data.

There are relatively few pairs of contexts such as that above, which produce unifiable but not identical contexts. More often, different contexts make either the same predictions for a given path, or contradictory predictions. Therefore, when two upper bounds are unifiable, it is quite often because they are identical. This entails that the elements in the unified output set of upper bounds are frequently the same as those of the input set. But where there is a difference at all, it is in the direction of greater informativeness. That is, the process of unification produces sets of upper bounds that are monotonically non-decreasing in specificity.

7.1.2 Discerning Distinct Lexical Items

The question arises of what to do when the cross-unification of two sets of upper bounds yields the empty set. In this case, the most reasonable conclusion is that the learner has encountered two unrelated senses of words with the same surface forms. These are stored as separate lexical entries, each with a lexical version multi-space representing a set of hypotheses about its feature structure.

For example, if the system did not contain a lexical entry for "train," and encountered instances of train as a verb (in the sense of 'educate') and as a noun (in the sense of 'locomotive'), the desired behavior would be for it to fail to find one feature structure which fit both pieces of input, and then to maintain two separate partial lexical entries, with future input adding further information to one or the other, but not both.

Of course, real-world input data may not immediately show up the need for separate lexical entries, or indicate which of two existing entries a new instance should belong to. In the above example, the system may encounter input which leads to the hypotheses that "train" is either a noun or an adjective in one case, and either a verb or an adjective in the other. This would lead to the incorrect conclusion that "train" must be an adjective.

The system is therefore susceptible to error from misleading data, which should not be surprising. Human language learners can also encounter new words in deceptive contexts,

13-

leading them to incorrect conclusions about the syntactic category of the word. In order for people to be able to acquire new words from context, however, such situations must be the exception rather than the norm, and will rarely be encountered in actual data. We assume that this is the case for the contexts encountered by our learning system as well.

Given such assumptions, incorrect conclusions such as that given above, while not advancing the knowledge of the system, will not be a serious hindrance to it, either. With further data containing the word in question, unless it occurs in similarly unusual contexts, the incorrect hypothesis will not lead to any successful parses. The learning procedure will therefore be invoked to learn the word again, generating hypotheses appropriate to the new contexts. At worst, therefore, the lexicon may contain some incorrect hypotheses that are not subsequently used, along with the correct hypotheses.

7.2 Combining Lower Bounds

For each pair of lexical version spaces whose upper bounds unify, there is a unified upper bound in the output hypothesis set. The set of lower bounds corresponding to this upper bound is the combination of the two sets of lower bounds corresponding to the two input upper bounds. The method of combination here involves comparing each member of one set with every member of the other, as above.

However, the pair-wise comparison of lower bounds does not involve unification. Almost any given pair of lower bounds contain some inconsistent information and thus fail to unify. This is because lower bounds generally include very specific information about other elements in the tree of the successful parse, which are distinct for almost any successful parses of different pieces of input. For example, the instantiation of a verb in a successful parse may include number and gender information about its objects, which is not generally part of the definition of the verb.

Rather, in comparing the two lower bounds, the system determines what information they have in common. As with combining the upper bounds, the goal is to determine a single feature structure for the new word that simultaneously satisfies the constraints of multiple pieces of input. For lower bounds, this means finding the most informative possible feature structure all of whose constraints are consistent with all of the instantiated parses licensed by that feature structure. Any constraint which is satisfied by one, but not the other, piece of input cannot be part of the definition of the word. Therefore, the output lower bound for a given pair of input lower bounds must contain only those feature paths which are in both of them.

The process of finding the shared information in two lower bounds is performed by the prolog procedure `cross_fs`. This is a recursive procedure, and relies on the fact that the paths within a feature structure are listed in alphabetical order.[2] To find the shared information of two feature structures, their first features are compared. If they are not identical, that which comes first alphabetically is discarded, and the comparison is repeated. If they are identical, the procedure then compares the values of this feature. If they are distinct atomic values, or complex feature values which themselves share no paths, then this path is again discarded, and comparison continues. If the values do have some information in common, then the path is added to the feature structure of the output lower bound, and the procedure is called recursively to determine the information which is shared by the values of this path in the two inputs. The complete code for this procedure is given in Appendix D.

The output of this process is a set of combinations of input lower bounds, which are now candidates to be lower bounds of the updated lexical version space. Several checks are then performed to determine whether each candidate should be added to the set of lower bounds corresponding to the updated upper bound. First, if the information shared by two lower bounds is equal to that in the upper bound,[3] then the combination of these lower bounds adds no information to the upper bound. Such a lower bound can be eliminated from consideration. Second, some of these pairwise comparisons may produce the same shared information, so the procedure checks for duplicates in the result of the comparisons. Finally, some of the results of the pairwise comparisons may produce a candidate lower bound which subsumes other such results. That is, a comparison may produce a lower bound which is identical with a node elsewhere in the lexical version space, between another, more specific lower bound and the corresponding upper bound. In such a case, the more general need not be maintained as a separate possibility, as it is already in the lexical version space as a possible definition of the word.

---

[2] UNICORN itself also depends on feature paths being listed in alphabetical order, so this reliance is not unique to the learning mechanism.

[3] The information shared by two lower bounds of a common upper bound can never be less than that of the upper bound, since they are both extensions of it.

The intuition here is that the correct definition of the word must be intermediate in informativeness between one of the hypothesized upper bounds and one of the corresponding lower bounds. That is, if a given lexical version space contains the correct definition, then the correct definition may be equivalent to any of the lower bounds, or it may be more general than any of them. When we have more than one encounter with a word, the correct definition must be consistent with at least one of the successful parses found on each encounter. But we do not yet know which of the successful parses represents the actual definition of the word. By looking at the information shared by two successful parses, we can see what the definition of the word would have to be in order for this pair of parses to have been produced. Some of these comparisons are uninformative, and their shared information is eliminated.

Unlike the case with the upper bounds, here there is no guarantee that each member of one set will only successfully combine with one member of the other, so the set of lower bounds in the output may well be larger than those of the inputs. In practice however, the size of the set of lower bounds generally decreases as more input is examined. This is due to the fact that, as input-specific information is abstracted away, the shared information among a number of pair-wise comparisons is likely to contain a number of duplicates, as well as candidate lower bounds which are in a subsumption relationship with other candidates.

7.3 An Example Learning Episode

This section gives a detailed description of a learning episode using the procedure described in the previous two sections. In particular, I describe a sample learning episode of the system, combining the constraints from two pieces of input to acquire the new word "glarf." The first piece of input encountered is the sentence (7.3). This example was discussed in order to illustrate the working of the system with a single piece of input in the previous chapter, and will be quickly reviewed here.

(7.3) John glarf left.

Having parsed "John" as a subject NP, the current test grammar predicts that a VP may come next.[4] It therefore predicts two possible syntactic categories for "glarf," adverb and verb, as these are the categories which can begin a verb phrase. The restrictor used for this example distinguishes verbs according to how many complements they take, but not according to the syntactic category of each of those complements. Therefore, two upper bounds are constructed for the syntax of the lexical entry of "glarf." One contains the information known by the system about adverbs, the other that which is known about verbs. For example, the system knows that all finite verbs must take subjects that are nominative case.

---

[4] There are other possibilities predicted, such as conjunction. However, these do not lead to any correct parses, and are therefore ignored here.

The lower bounds contain more detailed information. For the upper bound which assigns "glarf" to the category of adverb, there is a single lower bound with all of the feature structure used in producing a parse structurally isomorphic to that of (7.4). However, corresponding to the upper bound with "glarf" as a verb, there must be at least two separate lower bounds; one yielding a parse structurally similar to that of (7.5), the other to that of (7.6).

(7.4) John quickly left.

(7.5) John turned left.

(7.6) John had left.

The lexical version multi-space constructed from this piece of input, given in Figure 6.3, is repeated in Figure 7.2.

The next piece of input is sentence (7.7). This yields the same two upper bounds as the previous input, adverb and verb. Therefore, the set of upper bounds remains the same. As with the previous input, the upper bound in which "glarf" is an adverb leads to a successful parse, so the elements in the two sets of lower bounds are unified. Since there is just one element in each of the input sets, the resulting updated set of lower bounds also has one element, containing the information which the two input instantiations have in common. For example, while the lower bounds on the two inputs specify the semantics of the verb modified by the hypothesized adverb "glarf," these specifications contradict each other, and so this information is unspecified in the resulting lower bound.

(7.7) John glarf stopped.

In the case where "glarf" is a verb, the only successful parse with this input is one in which "stopped" is a past participle, since "stopped" cannot be an adverb, as "left" can. Therefore, we have a set with one element to do pairwise comparisons with the previous corresponding

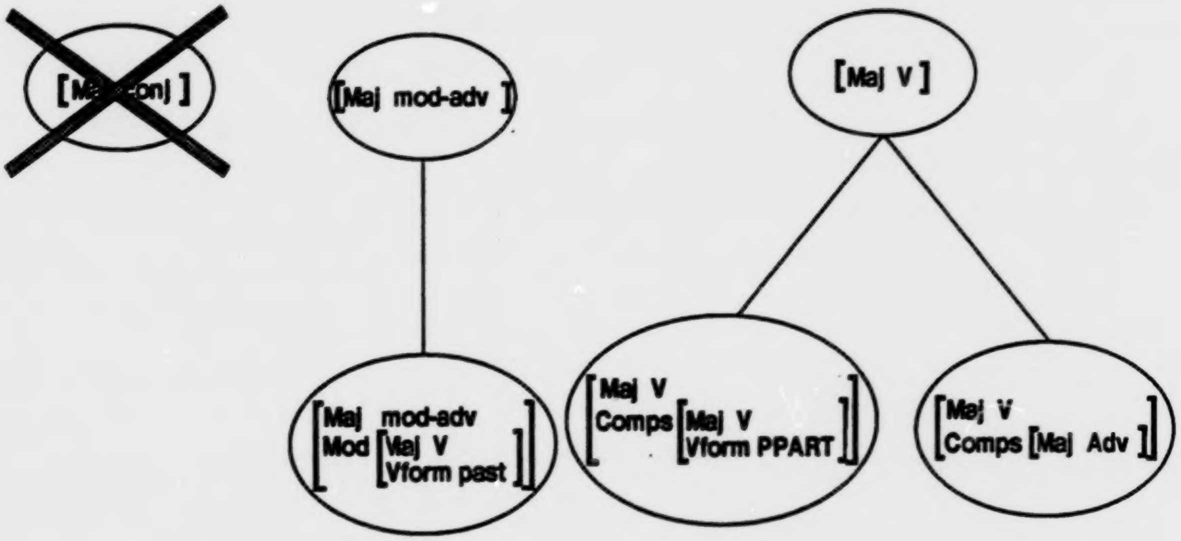
[Maj conj]
[Maj mod-adv]
[Maj V]
Maj mod-adv
Mod Maj V
Vform past
Maj V
Comps Maj V
Vform PPART
Maj V
Comps [Maj Adv]

Figure 7.2: Lexical version multi-space for "John glarf left."

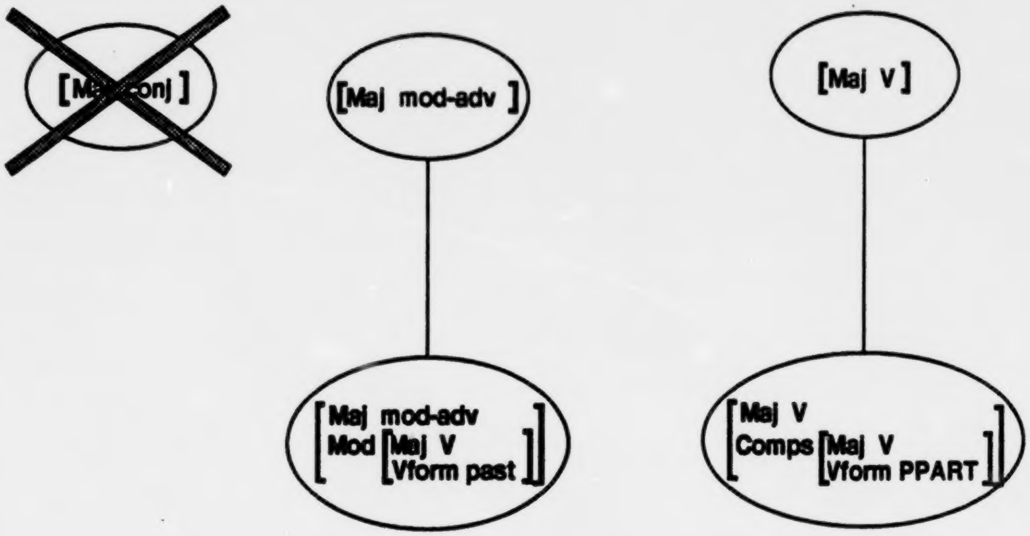

Figure 7.3: Lexical version multi-space for "John glarf stopped."

set with two elements. Comparing the two uses of "glarf" as an auxiliary verb yields a more generalised version of the same analysis, as with the adverb hypothesis. However, comparing successful parses of "glarf" as an auxiliary verb with one piece of input and as a verb with an adverb complement with the other yields no information beyond that of being a verb, which is already present in the upper bound. Therefore, this possibility is eliminated.

7.4 Convergence on a Lexical Definition

Complete convergence on the correct definition of a word occurs when only a unique lower bound remains at some point during a learning session, and it is equivalent to the unique upper bound. This indicates that the system has deduced the complete feature structure representation of the new word. At this point, the word may be considered to have been completely learned.

It is possible, and even likely, that such complete convergence will never occur, particularly if a given word is only rarely encountered. It then becomes a matter of extra-grammatical heuristics to decide when a lexical item has been completely learned. Such heuristics have not been implemented in the current learning mechanism, but in this section, I briefly discuss one or two of them.

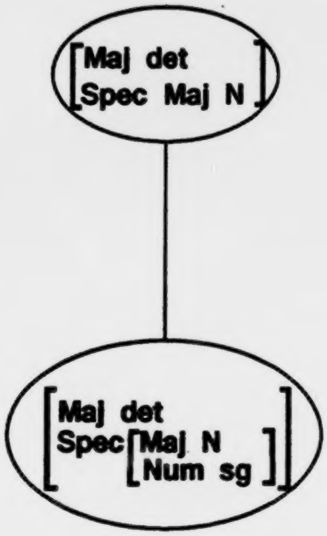

Figure 7.4: Lexical version space for a new determiner.

One of the reasons for lack of convergence in the system is that we do not use negative examples. That is, the system is presented with unknown words in example sentences, all of which are assumed to be grammatically correct and semantically meaningful. There is ample evidence that language learning in humans also makes use only of positive examples. However, this prevents some kinds of hypotheses from ever being excluded from consideration.

This is the case, for example, with the simple lexical version space for a new determiner, given in Figure 6.2, and repeated as Figure 7.4.

Here there are two possible hypotheses. The new word may either be a determiner which does not specify the number of the noun phrase it selects for, or it may be one which selects for a singular noun phrase. The lexical version space in Figure 7.4 is consistent with input where the determiner occurs with a following singular NP. If any input is seen in which the determiner is followed by a plural NP, then the lower bound will be generalized upward, so that it converges with the upper bound, yielding a unique hypotheses for the definition of the word.

However, suppose that no such input is ever seen. If the determiner is followed by a singular NP in all subsequent inputs, then there is no way to either further specialize the upper bound, or further generalize the lower bound, so the system will never converge on a unique solution.

145

Intuitively, if, after a reasonable number of encounters with a variety of input data, the lower bound is not raised, the system should assume that this is because the lower bound is, in fact, correct. This intuition should be encoded as a heuristic for determining that a word has been learned, in spite of the fact that the procedure has not converged to a single node of a lexical version multi-space.

Therefore, when the hypothesis space for a new word consists of a single lexical version space, with one upper bound and exactly one lower bound, after a reasonable range of contexts in the input, the system should conclude that the lower bound represents the correct definition. Further work, including empirical studies, will be necessary to determine what constitutes a reasonable number and variety of pieces of input to make this determination for a given word.

The procedure we have described enables the parser to make use of context from multiple pieces of input in order to infer the syntactic feature structure of newly encountered lexical items. While it is susceptible to error from misleading presentations of data, such presentations are not the norm, and merely lead to the necessity of further input before the correct target lexical items are learned. The result of a learning episode is often not complete convergence, so the learned lexical items may be viewed as less than fully specified.

7.5 Computational Complexity

In this section, I analyse the time and space complexity of the learning algorithm, and its influence on the complexity of the UNICORN system within which it operates. Both the complexity of the acquisition algorithm and that of Earley's algorithm vary according to the type of grammar they are used with. I will therefore discuss complexity both in the general case, and with the particular class of grammars with which the system has been tested. I first discuss the issue of time complexity, and then that of space.

7.5.1 Time

This analysis is given in terms of *n*, the number of words in the input sentence. According to [Earley, 1970], Earley's algorithm parses input of length n in a time bounded by n^3 in the general case. I here provide an analysis of the time complexity of the learning mechanism in terms of the number of primitive steps that must be executed, proportional to the length of the input.

For the most general case, the length of the input is taken to be n. However, the actual relevant measure is m, the length of the string from the unknown word to the end of the input, where $m \leq n$. Therefore, the worst case of complexity occurs when the unknown is the first word of the sentence.

The first step of the algorithm is to start a learning session. This involves initializing a new file to hold lexical version multi-spaces for hypothesized definitions of new words, and requires only constant time. This initialisation is performed only once, before any input is parsed. The first of the steps performed on each encounter with an unknown word is to create a list of candidates, possible upper bounds for the definition of the word. This involves looking at $\sim n$ state sets, each with $\sim n$ states,[5] and constructing a single set of distinct states with no duplicates. This requires time $\sim n^4$, and yields a set of size $\sim n$.

For each of these $\sim n$ candidates the algorithm will then

1. Construct a lexical grammar for the candidate
2. Temporarily assert that lexical grammar
3. Continue parsing to instantiate the lexical grammar
4. Retract the lexical grammar

Constructing the lexical grammar requires time proportional to k^2, where k is the number of feature-value pairs in the type of the candidate. Temporarily asserting the grammar and later retracting it take constant time. To continue parsing, the system calls Earley's algorithm with each candidate, and continues as before. As discussed by [Earley, 1970], Earley's algorithm has a time bound of n^3.[6] Since this must be done for each candidate, the overall complexity of trying the candidates requires time $\sim n^4$.

After all of the candidates have been instantiated, the system then combines the newly established lexical version multi-space with the previous lexical version multi-space for that word. The complexity of combining version spaces is discussed by [Hirsh, 1990], and his analysis must be modified only slightly in order to provide the complexity of combining lexical version multi-spaces.

---

[5] These figures are taken from the complexity analysis of Earley's algorithm given in [Earley, 1970].

[6] With unambiguous grammars, Earley's algorithm operates in time $\sim n^2$, and so with the addition of the learning mechanism, the entire system would run in time $\sim n^3$. However, the HPSG-style grammars currently being used with the system are not unambiguous.

The first step in combining two lexical version multi-spaces is taking the cross-unification of the two sets of upper bounds. In the worst case, if there are a total of u_1 upper bounds in one lexical version multi-space and u_2 in the other, there are at most $u_1 u_2$ upper bounds in the combined, updated lexical version multi-space. Each comparison has the complexity of one unification operation. It has been shown by [Knight, 1989] that this can be done in time proportional to k^2, where k is the number of feature-value pairs in the types being compared. The overall complexity is thus proportional to at most $u_1 u_2\ k^2$.

In combining the sets of lower bounds of two lexical version multi-spaces, the greatest number of pairwise comparisons between lower bounds must be done when there is only one version space in each multi-space; otherwise, the lower bounds from one version space are compared only with the lower bounds of one other version space, that whose upper bound unifies with its own. Thus in the worst case, if there are a total of l_1 lower bounds in one lexical version multi-space and l_2 in the other, there are at most $l_1 l_2$ lower bounds in the combined, updated lexical version multi-space.

In each node, feature-value pairs are tree-structured, as described in Chapter Three. According to Hirsh's analysis, the algorithm for determining the feature structure information shared by two nodes whose feature-values are tree-structured takes time proportional to $k \log^2 v$, where k is the number of feature-value pairs in the types being compared, and v is the maximum number of features any value can take. Determining the complete set of lower bounds thus takes time proportional to $l_1 l_2\ k \log^2 v$.

Eliminating those elements of the updated set of lower bounds that are equivalent to the unique upper bound requires an additional $l_1 l_2\ k \log^2 v$, while eliminating those which subsume some other member of this set requires time proportional to $(l_1 l_2)^2\ k \log^2 v$. The overall complexity of combining the two sets of lower bounds is thus proportional to at most $(l_1 l_2)^2\ k \log^2 v$.

Therefore, the total time for combining two lexical version multi-spaces is $u_1 u_2\ k^2 + (l_1 l_2)^2\ k \log^2 v$. In practice, the latter term always predominates, for two reasons. First, the cardinality of $l_1 l_2$ is necessarily as large or larger than that of $u_1 u_2$, since each upper bound must have one or more corresponding lower bounds. In fact, as discussed in Section 7.1, the size of the term $u_1 u_2$ is in practice rarely larger than the smaller of the two terms u_1 and u_2, because the operation of cross-unification has the effect of intersection when it is performed on input sets whose members are non-unifiable with each other. Second, the subset of k which actually occurs on an upper bound is necessarily a subset of that which occurs on its corresponding lower bounds, since the upper bound always subsumes the lower.

The overall time complexity of the learning system is therefore bounded by n^4 or $(l_1 l_2)^2 k \log^2 v$, whichever is greater. As mentioned above, n here actually represents not the total number of words in the input, but the number of words following the point of failure upon reaching the unknown. While this number is equal to n in the worst case, it may be as little as zero, when the unknown is the last word of the input. When this number is small, then the latter term predominates. However, when the unknown occurs more than a few words from the end of the input, the n^4 term predominates, providing an upper bound for the overall time complexity of the system.

7.5.2 Space

Upon failure in parsing, the space is taken up by $\sim n$ state sets, each with $\sim n$ states. Each state set represents a different way of parsing the input up to that point, and each state within a state set represents a distinct set of predictions regarding the next lexical item of the input. In parsing, this entails that the space bound is $\sim n^2$ in general, as described by [Earley, 1970].

However, the learning mechanism does not maintain space for duplicate lexical entries resulting from predictions made in distinct state sets. For example, if there are two distinct ways of parsing the initial prefix of the input up to the point of the unknown, both of which make the prediction that the unknown may be a noun, this possibility is stored only once. This entails that the total number of distinct predictions stored is $\sim n$. This is therefore also the number of upper bounds to be stored, and the number of distinct lexical version spaces in a lexical version multi-space.

For each upper bound, there may be $\sim n$ possible ways of instantiating it by parsing the remainder of the input, to yield $\sim n$ lower bounds. The space bound of a lexical version multi-space is therefore $\sim n^2$.

The incorporation of the learning mechanism into UNICORN does not increase the space complexity bound of the system for parsing, although it does greatly increase the associated constant. In general, for any given state set obtained at a given point in the parse, the scanner finds only a few lexical items that fit the constraints imposed by the possibly large number of predictions. The learning mechanism, however, must construct a lexical hypothesis for each distinct set of predictions. This is equivalent to worst-case behavior for the parser, the case in which there exists a lexical item that fits each set of predictions in the state set, and the correct lexical item must determined by seeing which leads to success in parsing the rest of the input.

The space complexity of the system which incorporates the learning mechanism is at most proportional to n^2, just as it is without the learning mechanism. However, the behavior with the learning mechanism is consistently close to the worst-case requirements for space complexity, while without the learning mechanism it rarely approaches this bound.

Chapter 8

Related Issues

In this chapter, I discuss some issues which are outside the focus of this dissertation, but which are related to it. I first address the issue of lexical rules. While these have been implemented in the system in a limited way, I have not yet made full use of their power. Lexical rules are an extremely powerful, but computationally expensive mechanism for parsing and acquisition, and the current hardware does not have the speed or memory required to make a wider application of them practical. Nevertheless, I provide a sketch of the kind of treatment available with lexical rules for a range of phenomena, including metonymy, metaphor and the use of a word of one part of speech as another.

I also address the question of storing the newly learned information as part of the permanent lexicon of the system. This relates to the issue of generation, which I have not previously discussed, as the current system deals solely with understanding. However, the system described here has interesting implications for a generator, which I discuss briefly.

8.1 Lexical Rules and Their Applications

Lexical rules have been postulated in one form or another in a number of grammatical theories, most notably Lexical Functional Grammar ([Bresnan, 1982]), GPSG ([Gazdar *et al.*, 1985a]), and more recently, HPSG ([Pollard and Sag, In press]). The motivation for lexical rules is to achieve the effect of a very large, possibly infinite lexicon, without storing a large or infinite number of distinct and unrelated lexical items. Lexical rules capture generalizations among lexical items which are systematically related. Each rules thus licenses one lexical item, which is not stored separately in the lexicon, on the basis of another, which is itself either stored in the lexicon or licensed by another lexical rule.

There are a number of applications of lexical rules which I discuss in this section. Some, such as their use in relating the various morphological forms of a word, have been discussed previously in the literature. Others, such as their possible use to implement referring functions, are the subject of ongoing research.

8.1.1 Relating Morphological Forms of a Word

The first purpose for which lexical rules were proposed (in [Bresnan, 1978]) was to account for the alternative subcategorisation frames with which various forms of a word occur, and thus provide an analysis of, *e.g.*, passive forms that does not require NP-movement rules. Lexical Functional Grammar (LFG) thus proposes a rule to license the passive form of a verb on the basis of its corresponding active form.

This is in contrast to earlier transformational treatments of passivisation. In standard Transformational Grammar (TG, [Chomsky, 1965]), passivisation is viewed as a process which derives one structure from another. A lexical rule is not a process or a derivation, but a means of determining that a lexical item is well-formed, by reference to another lexical item.

It is therefore perhaps misleading, though well entrenched in tradition, that lexical rules are written with an arrow, with the licensing form on the left and the licensed form on the right. For example, the HPSG version of the lexical rule for licensing passive forms is given in (8.1) (from [Pollard and Sag, 1987], p. 215).

(8.1) **Passive Lexical Rule**

$$\text{PASSIVE:}\quad {}_{\mathit{base}\wedge\mathit{trans}}\begin{bmatrix} \text{PHON } \boxed{1} \\ \text{PAST}-\text{PART } \boxed{2} \\ \text{SYN|LOC|SUBCAT} < \ldots, [\ ]_{\boxed{3}}, [\ ]_{\boxed{4}} > \\ \text{SEM|CONT} \boxed{5} \end{bmatrix} \mapsto$$

$${}_{\mathit{passive}}\begin{bmatrix} \text{PHON } f_{PSP}(\boxed{1}, \boxed{2}) \\ \text{SEM|LOC|SUBCAT} < (\text{PP[BY]}_{\boxed{4}}), \ldots, [\ ]_{\boxed{3}} > \\ \text{SEM|CONT } \boxed{5} \end{bmatrix}$$

Here the term f_{PSP} in the PHON feature to the right of the arrow is used to denote the phonological form which represents the application of the 'past participle' function to the phonological form of the lexical item, along with any information given about how its past participle is formed, both given on the left side of the arrow. For other feature paths as well, information on the passive form is determined on the basis of specified paths on the input form.[1]

More generally, lexical rules are used to relate the forms of an inflectional paradigm, for example, capturing facts about the morphology of agreement patterns of English verbs. For a language such as English, with a modest number of distinct morphological forms for any given word, capturing such generalisations is convenient, but not absolutely necessary for an NLP system. It would be possible to construct a lexicon such that each verb had half a dozen distinct and unrelated lexical entries. In fact, in the absence of an implementation of lexical rules for UNICORN, this is what I have done for the system described here. Each morphological form has its own entry in the lexicon, and there is nothing to indicate that these forms are, in some sense, the same word, except that they contain the same values for some of their paths, primarily in their semantics.

---

[1] As noticed by Georgia Green, there is some theoretical difficulty in the use of structure-sharing to relate forms on the left and right side of a lexical rule. The intuition is that the output form should have feature values which are related in a principled way to those on the input form. However, specifying a relationship of structure-sharing between the two entails that once a value for such a feature path is instantiated for the output form, it must be instantiated the same way on the input form. Thus, the application of one such rule determines features on the lexical entry of the input form which may be idiosyncratic to the context in which an instance of the output form occurs.

However, using lexical rules to encode morphological paradigms becomes a necessity for languages with more robust morphological inflection. For example, in Cheyenne, an Amerindian language of the Algonquian family, each verb is inflected for agreement not only with its subject, but with its direct and indirect objects as well. The language is polysynthetic, so that there is not a one-to-one correspondence between agreement morphemes and feature-values being marked. Rather, a given morpheme may indicate some combination of person, number and animacy features for any one, two or three of its participants. In addition, there are a number of aspects and moods, each of which has its own morphological marking, and interacts with the agreement inflection.[2]

Even for a language such as English, there are limitations in the abilities of the learning mechanism, due to the lack of lexical rules. At present, each morphological form of a word must be acquired as a distinct lexical item. For example, if the system learns from context the form of a verb which is in the present active indicative, and has a third person singular subject, it will fail to understand an occurrence of the same verb in the present active indicative with a third person plural subject. This must be learned from scratch, in exactly the same way as a completely unrelated word.

To some degree, this lack of generalization has been deliberate. Along with generalizations about morphological paradigms comes the risk of acquiring incorrect information, in the case of words whose pattern is irregular, but not known to be so. By acquiring each morphological form separately, the system fails to represent elegantly the information shared by the various forms of a word. At the same time, however, it ensures that those forms which actually are acquired are in fact correct.

Heuristics based on known morphological paradigms would be valuable for providing reasonable guesses for what the related forms of a newly learned word should be. For example, a new verb ending in "-ind" will very likely have a past tense ending in "-ound," as is the case with "find," "bind," "grind," and "wind." However, such heuristics are fallible. If this heuristic is now used in the acquisition of the verb "mind," its past tense will be incorrect.

---

[2] For a formal description of the morphological agreement pattern within the framework of GPSG, see [Russell, 1987].

An exploration of the proper use of such heuristics along with lexical rules to capture generalisations about morphological paradigms is a topic for future work. The current system lacks both the elegance and the vulnerabilities of such heuristics.

8.1.2 Metonymy

Metonymy is the act of referring to a given entity by mentioning some other entity which is associated with it. It is commonly called a figure of speech, but it occurs quite frequently in very ordinary language. A wide range of examples of metonymy are discussed by Nunberg in his dissertation, [Nunberg, 1978].

The association between the entity referred to and the one mentioned can be conventional and widely known, such as uttering "the White House" to refer to the president of the U.S. Other kinds of metonymy are generated by systematic rules, so that they are easily recognizable in context, although the particular instance of it may not have been encountered before. Such metonymy includes uttering the name of the capital of a state or country to refer to the government of that state or country, the name of an author to refer to their works, or the name of an animal to refer to the meat or other commodity obtained from that animal. Some examples are given in (8.2).

(8.2) a. Relations between Paris and Bonn have been increasingly strained recently.
b. I had to read some of Milton in an undergrad humanities class.
c. Everyone says that brontosaurus tastes just like chicken, but I've never tried it.
d. Armadillo is supposed to make the longest-lasting shoe leather, but rattlesnake looks a lot nicer.

Synecdoche, also commonly referred to as a figure of speech, is a particular subtype of metonymy with a systematic kind of association. Here the association is that of a sub-type to a super-type.

Still other kinds of metonymy involve an association which is entirely dependent on the real-world context. For example, to take Nunberg's well-known example, a waitress can refer to a customer by mentioning a menu item which he has ordered, as in (8.3).

(8.3) The ham sandwich at table 6 wants some coffee.

So there is a sense of "ham sandwich" which refers to a human diner. However, this sense should not be stored in the lexicon, as it is not an idiosyncratic lexical property, but an instance of a more general pattern.

For the more conventionalised of these examples, it could be argued that they are best handled by expanding the lexicon. For example, there could be two entries in the lexicon under "chicken," one denoting the bird, the other the meat obtained from the bird. However, Nunberg gives a number of arguments against this approach. First, referring to a kind of meat by mentioning the animal it comes from works almost equally well with animals that are not commonly eaten, as seen with "brontosaurus" in (8.2)c.[3] This is the case with many metonymic associations.

Second, there are a large number of such associations which can form the basis of metonymy; given appropriate contexts, there is likely no principled limit. Therefore, a lexicon which included all metonymic senses would be not just very large, but potentially infinite.

The solution proposed by Nunberg is to formulate **referring functions**, productive rules which take one sense of a word as input and have a different sense as output. So, for example, there is one referring function that maps a name of a person onto the literary works composed by that person, and another which maps a menu item onto a person who has ordered that item.

Another case in which referring functions would be useful would be in the mass/count distinction. This was discussed in Chapter Two, where it was shown that assigning a lexical item the feature value MASS or COUNT irrevocably, either by querying the user or by asking for judgments on example sentences, is inadequate. This is because the vast majority of words can be used in both a mass sense or a count sense, depending on the context. This does not entail that they are unspecified for this distinction, since in most contexts, only one or the other can be used felicitously. Rather, the mass/count distinction is tied to the sense, which in turn may often be licensed by a referring function. For example, in the earlier polysemy between an animal and the meat obtained from it, the former functions as a count noun, the latter as a mass noun.

---

[3] This is true even with animals for which separate lexical items exist for the animal and its meat, such as "cow" and "beef." Here, the specialised term for the meat takes preference over the term licensed by the referring function. However, utterances such as (i) are still grammatical, and are appropriate in certain contexts.

(i) I want a healthy serving of grilled cow on a sesame seed bun.

Storing mass and count senses of words as unrelated lexical items might be preferable in some cases, such as "country", some of whose mass and count senses are only indirectly related, as seen in (8.4).

(8.4) a. West Virginia has some real pretty country.

b. Western Europe has some real pretty countries.

However, this is not acceptable as a general solution. As shown in the examples discussed in Chapter Two, repeated as (8.5) and (8.6), many words that are generally assumed to have one value for the mass/count distinction can take the opposite value in an appropriate context. This would entail almost doubling the number of nominal entries in the lexical, encoding both a mass form and a count form for most common nouns. More generally, examples such as (8.3) show that without rules to capture these derived senses, a learning mechanism could acquire an infinite number of new senses.

(8.5) It smells like new baby in here.

(8.6) Glucose, sucrose and fructose are the only sugars I know of.

Referring functions bear a clear similarity to lexical rules. In both cases, the rule should not be thought of as a procedure for turning one lexical item into another, but rather as a way of licensing one lexical item on the basis of another. Therefore, lexical rules should provide a straightforward means of implementing referring functions.

Before this can be done, however, the representation of referring functions and the contexts which they make reference to must be formalised. This is the topic of ongoing work by Steve Helmreich [Helmreich, 1985].

8.1.3 Metaphor

At this point, I do not have much to say about metaphor as a separate phenomenon. In a sense, metaphor can be regarded as a kind of metonymy, where the association is simply that of sharing some contextually relevant property. As with metonymy, therefore, the most promising approach to metaphor is to use referring functions.

In the work of Guo [Guo, 1987], a metaphorical usage comes from relaxing the semantic preference of a predicate. This is sufficient for Guo's purpose, which is to recognize occurrences of metaphor, so that by process of elimination, the need for learning new word senses can be detected. However, this is not by itself sufficient for determining the actual referent of a metaphorical usage. For this, is is also necessary to determine which of the semantic restrictions are maintained and which are relaxed to relate the literal meaning to the idiomatic. Such an approach to metaphor is outlined in [Martin, 1988]. A further project for future work is to integrate this treatment of metaphor into my learning mechanism.

8.1.4 Learning New Syntactic Categories for Previously Known Words

One principled reason for the incompleteness of any lexicon is that words of one syntactic category can productively be used in another syntactic category. Even with a supposedly complete lexicon, in which the lexicon contains all the words in the input, parsing still fails in cases where a word which is known in one syntactic category is being used as another. This is the phenomenon which I discuss in this section.

There are two subcases of words of one syntactic category being used as another. In the first, there is overt derivational morphology, so that it is obvious from the form of the word that it is a derived form. Its syntactic category is unambiguously indicated by the derivational affix. Several such cases have been implemented in the current grammar, and are described in Section 8.1.4.1.

In the other subcase, a word of one syntactic category is used as another with no morphology to signal that this has happened, although such morphology is in some cases available. An analysis of this phenomenon has not been implemented in the current system, but a treatment is sketched in the remaining subsections of this section.

8.1.4.1 Morphologically Derived Categories

This section addresses those forms in which a derivational affix signals the syntactic category of the word. The term 'derivational' implies that the word has undergone a procedure, that a word of one syntactic category has been changed to another. In keeping with the principles of HPSG, however, derivational affixes are instead viewed as licensing one lexical form on the basis of another. This again is accomplished by means of lexical rules.

15-

In a system which implemented lexical rules directly, the effect of such rules would not be visible in the syntactic phrase structure markers of sentences. For example, there would be a lexical rule to derive gerunds from base form verbs, with roughly the following form.

(8.7) Gerund Lexical Rule

$$\text{GERUND:} \quad \begin{bmatrix} \text{PHON } \boxed{1} \\ \text{GERUND } \boxed{2} \\ \text{SYN|LOC|SUBCAT} < \boxed{3} > \\ \text{SEM|CONT} \boxed{4} \end{bmatrix}_{base} \mapsto$$

$$\begin{bmatrix} \text{PHON } f_{GER}(\boxed{1}, \boxed{2}) \\ \text{SEM|LOC|SUBCAT} < \boxed{3} > \\ \text{SEM|CONT } \boxed{4} \end{bmatrix}_{gerund}$$

In this rule, the output form has exactly the same SUBCAT list as the input. So, for example, the rule licenses a gerund form of "head" which, like the verb form, subcategorizes for a nominal subject and an adverbial complement. The output form occurs in (8.8), where it is saturated for complements, but unsaturated for subjects.[4]

(8.8) Heading toward the interstate now would be a good idea.

In the absence of a method for encoding lexical rules directly in Unicorn grammars, I have mimicked the effect of lexical rules by syntactic rules, so that the derivation of morphologically derived forms is visible in the phrase marker of the sentence. The syntactic rule that does the work of (8.7) in the current grammar is shown in (8.9), and the resulting phrase marker for (8.8) is given in Figure 8.1.

(8.9)

```
%% Gerund Rule
n -> v:
<x0 phon> = <x1 phon>
```

---

[4] I am assuming that gerund subjects are left syntactically unspecified, with their referent supplied by pragmatics. I do not address this issue here.

```
<x0 dtrs unarydtr> = <x1>
<x0 synsem local category head maj> = <x0 maj>
<x1 synsem local category head form> = prog
<x1 synsem local category head case> = acc nom
<x0 synsem local category comps> =
   <x1 synsem local category comps>
<x1 synsem local category subj rest> = end
<x0 synsem local category bar> = two
<x1 synsem local category bar> = two
<x0 synsem local category marking> = unmarked
<x0 synsem local conj> = <x1 synsem local conj>
<x0 synsem local content> = <x1 synsem local content>
<x0 synsem local context> = <x1 synsem local context>.
```

Notice that the current encoding of this rule is not lexical. That is, the relation between the input and output forms is between phrases, rather than words. There is nothing in the grammar to prevent these rules from being formulated purely lexically. However, this would necessitate the addition of some other syntactic rules which are not currently needed in the grammar. For example, if the gerund rule were reformulated as a lexical rule, this would require the addition of a rule to combine a lexical noun with its complements. As there are currently no other cases where such a rule is required, it is more efficient to write the grammar in such a way that this rule does not have to be added.

However, if such rules come for free as a consequence of more general rule schemata, there is no additional cost in adding them. This is a topic for future investigation. For now, I merely note that the above rule is not intended as a complete analysis of gerunds, but only as a way of mimicking the effect of lexical rules in a system which does not yet allow them to be encoded directly.

8.1.4.2 Productive Non-Morphological Category Changing

In this section, I discuss the uses of a word belonging to one syntactic category as an element of another, without any overt derivational morphology. At first glance, this may seem to be a relatively rare occurrence, but in natural language texts, examples actually appear quite frequently.

```
               S
              / \
             /   \
           N2     V2
          / \       \
         /   \       \
       N2   ADJ2      V1
       |      |       /\
       |      |      /  \
       V2   ADJ1    V0   V2
       |      |     |     |
       |      |     |     |
       V1   ADJ0  would   V1
       /\     |           /\
      /  \    |          /  \
    V0  ADV2  now      V0    N2
     |    |            |     /\
     |    |            |    /  \
Heading ADV1          be  DET   N1
         /\                |    /\
        /  \               |   /  \
     ADV0  N2              a  ADJ1 N1
      |    /\                  |    |
      |   /  \                 |    |
  toward DET  N1              ADJ0 N0
          |    |               |    |
          |    |               |    |
         the  N0              good  idea
               |
               |
           interstate
```

Figure 8.1: Syntactic Phrase Marker for the sentence "Heading toward the interstate now would be a good idea."

Noun-noun modification is one case of the general phenomenon that has been extensively studied (see, for example, [Downing, 1977] and [Dowty, 1979]), but there are many others. For example, according to [Clark and Clark, 1980], any noun can be used as a verb, yet no on-line dictionary should list every noun as having a second entry in the lexicon as a verb. Therefore, a general treatment for words of one syntactic category being used as another is called for.

The phenomenon is not totally unrestrained; *i.e.*, it is not the case that a word of any syntactic category be used as any other syntactic category. A first approximation of the range of the process would be to say that it operates among all and only the open-class words, not closed-class (equivalently, content words, not function words). It maps only words of one open class into another open class.[5]

However, this is not quite correct, as shown by the chart in Figure 8.2. In each box of this chart there is an example in which the category listed on the left edge is used as the category listed along the top. Prepositions can be used as other categories, but I have not found an example of any other category can be used as a preposition.[6] I do not yet know the explanation for this asymmetry. In the current system, such cases are treated as entirely new words. Recall from Chapter Three that Earley's algorithm looks up a word in all and only those syntactic categories which have been predicted as possibilities for it. Therefore if the input word does not exist in the lexicon in any of those categories, it is assumed to be unknown, and is learned from context.

This is acceptable solution as a first approximation, in that it will eventually produce correct results. However, it fails to capture generalizations that may be made from the knowledge that a word already exists in the lexicon, but in a different syntactic category. I here explore briefly how such generalizations could be captured by expansions to the current system.

8.1.4.3 Alternative Treatments

In this section, I discuss alternative ways in which this phenomenon could be handled or explained away. First, it could be argued that each instance of what I have claimed to be one

---

[5] I am ignoring the case where a word of any syntactic category which is placed inside quotation marks and used as a noun. The referent of this noun is just the word itself, and is not derived from the referent of the quoted word; *e.g.* "You used a 'the' where you should have used an 'a'."

[6] It is often assumed that prepositions are a closed class, but there is evidence that this is incorrect. Unlike other closed classes, such as determiners, there are prepositions which many native speakers do not learn until well into adulthood, if at all. These include "via," "vis-a-vis," "re" and perhaps "versus."

BEST COPY AVAILABLE

	used as			
cat	noun	verb	adjective	adverb
noun	—	impact his students	elevator operator	surface observable
verb	this book is a great read	—	delete button	freeze dry
adj	the young and the restless	wilding in the park	—	circular breathe
adv			an almost arch	—
prep	the ins and outs	they outed the senator	my in box	burned through

Figure 8.2: Words of one syntactic category used as another.

syntactic category being used as another actually calls for an additional Phrase Structure rule. This would maintain the view that each lexical item is of a particular basic or inherent category, and a word cannot be used in other than its given category. However, syntactic rules may license the occurrence of a category in other than its normal syntactic environment.

I argue against this approach, and in favor of an approach using lexical rules, in which these words in new categories do not appear in syntactic trees nor in the lexicon, but are licensed on the basis of existing lexical entries.

The Syntactic Approach. One possible account of the phenomena shown in Figure 8.2 is that each entry of the chart motivates one or more new Phrase Structure rules. Such an account has often been claimed for noun-noun modification (see discussion in [Dowty, 1979], pp. 314-319). The new PS rule here is as in (8.10).

(8.10) N0 → N1 N0

The claim here, implicit in the phrase noun-noun modification, is that this is not a case of a noun that is an adjective, at any level of the Phrase Structure tree. Rather, it is a case of a noun that is used to modify another noun. Because prenominal adjectives are specified to modify nominal categories of bar-level 1, this captures the generalization that adjectival modifiers must precede nominal modifiers within an NP. This approach thus predicts the goodness of (8.11) and the badness of (8.12).

(8.11) pretty little red brick houses

(8.12) * red brick pretty little houses

Such a rule does not conflict with the principles of HPSG. All that is necessary is to specify that the first of the two nouns syntactically modifies the second. However, a similar approach is not feasible for all the entries of Figure 8.2. For example, the VP "impact his students" would require the PS rule in (8.13) and would have the phrase marker shown in (8.14).

(8.13) VP → N NP

(8.14) impact his students

```
       V2
       |
       |
       V1
      / \
     /   \
   N0     N2
   |       /\
   |      /  \
impact  DET  N1
         |    |
         |    |
        his  N0
              |
              |
           students
```

However, such a rule violates the Head Feature Principle (HFP) of HPSG (described in Chapter Three), which states that in any headed structure, the head features of the mother are structure-shared with those of the head daughter. Since the syntactic category is encoded as a head feature, a rule such as (8.13) cannot license a headed structure, as it must do if the left daughter is the head of the phrase.

In addition to (8.10) and (8.13), the examples in Figure 8.2 would require the addition of at least the following PS rules.

(8.15)a. ADJ → N ADJ, e. NP → DET ADJ, i. NP → DET P
b. NP → ADJ V, f. VP → ADJ ADV, j. VP → P NP
c. NP → V N, g. VP → ADJ V, k. NP → P N
d. V → V V, h. NP → ADV N, l. VP → V P

Of these, (8.15)a., c., d., g., h., k. and l. are similar to (8.10), in that no violation of the HFP. However, the other rules, (8.15)b., e., f., i. and j., all involve such violations. This argues strongly against taking such an approach to explain all of the data in Figure 8.2.

There are practical reasons for rejecting this approach as well. While no single rule of (8.15) is a serious hindrance to parsing by itself, the combined effect of a large number of such rules would be to make parsing extremely slow, by introducing a large number of spurious categorial ambiguities, which would have to be eliminated by the parser.

Further evidence against such an approach in the general case is provided by morphology. In examples such as "wilding" and "ins and outs," we see that words of a derived syntactic category take inflectional morphology of the type appropriate to the derived category, not of the input category.

Using Lexical Rules. Using lexical rules, the treatment of words of one syntactic category being used as another is in principle no different from that of words with derivational morphology. The only difference in the rule itself is that without derivational morphology, the PHON value of the output will be the same as that of the input.[7]

The real differences between forms with derivational morphology and those without are in detecting the situations where the lexical rule should apply. As mentioned in the previous section, with derivational morphology, the category of the output form is unambiguous. Without derivational morphology, however, the category is inherently ambiguous.

This ambiguity makes it non-trivial to identify the target of learning, in contrast with the case of completely unknown words. The fact that parsing fails although all of the words are found in the lexicon indicates that some word does not exist in the lexicon with the syntactic category which it would have to have in order for the parse to succeed. The difficulty is knowing which word's category to experiment with, and what to try for it instead.

This question will be the subject of ongoing work, but there are known heuristics which will be useful. First, it is generally possible to determine that a certain string of words cannot lead to any successful parse before the parser has examined all the words of the sentence. A reasonable first guess would be to start at the point at which parsing fails, and try other categories for the word immediately preceding that point. If that produces no good results, work progressively backwards from there.

---

[7] Equivalently, the PHON value of the output is the application of the identity function to that of the input.

The system should also make use of heuristics about how likely it is for certain categories to be used as other categories. For example, using an adjective as a noun is extremely common, while using an adjective as a verb is fairly rare.

In addition, the fact that not all syntactic categories participate in category shifting allows the parser to make some inviolable predictions. For example, knowing that a determiner cannot be used as any other syntactic category and that a determiner necessarily introduces a noun phrase leads the system to try to complete a noun phrase whenever it has seen a determiner. If the current partial chart does not complete the NP begun by a determiner, then the system should guess that a word of some other category is being used as a noun as in (8.16).

(8.16) First you need to do a delete.

Another question for future work is that of inferring the semantics of the derived category from that of the input lexical item, although it is possible to give a rough outline of how this question should be answered. With an unknown word, the system must construct the semantics from scratch. But here, the meaning of the word in its original category provides a reasonable hypothesis as to the meaning of the derived category. For example, in (8.17), the system must infer the meaning of the word "left" as a noun, assuming that its meaning as an adjective is known.

(8.17) You've got a left onto Elm at the next corner.

As with an unknown word, a partial representation is constructed from the known words in the sentence. The semantic content contains an unknown entity denoted by "a left," a path-change to the entity denoted by "Elm" and a location denoted by "the next corner." The domain-specific knowledge base has only a few actions which effect a path-change, such as 'turn' and 'bear.' No such action is explicitly mentioned anywhere in the utterance, but the selectional restrictions of "left" as an adjective allow it to modify either a turn- or a bear-action. The lexical rule which licenses the use of an adjective as a noun encodes a referring function, which states that an adjective may be used in a context where the content of the modified noun is inferrable, with the semantics of both the noun and the adjective. In this case, "left" is an adjective being used as a noun, with the meaning of either 'a left turn' or 'a bearing left.'

The final question regarding the use of lexical rules here is that of when they should apply in the parsing process. If every lexical rule applies at every point at which its conditions are met, this is equivalent to automatically expanding the lexicon to include all of the senses that a word might receive in appropriate contexts. However, as was seen in Section 8.1.2, there is no principled limit to the number of senses that a given word might take on in appropriate contexts. This approach is therefore not practical. In addition, it would lead to rampant lexical ambiguity, which would greatly slow down the parsing process.

The most promising alternative would be a failure-driven mechanism for invoking lexical rules. That is, a lexical rule would be applied only when there is a failure without it. Different lexical rules would be invoked for different kinds of failure. For example, a failure of syntactic parsing would point to the application of a lexical rule licensing a word in a new category. A failure of semantic checking, with a well-formed syntactic structure, would point to the application of a lexical rule encoding a referring function. Again, this will be a topic for future research.

8.2 Idioms

Idioms on the one hand and metaphor and metonymy on the other are similar, in that in both cases, input that is syntactically well-formed is rejected by the system, because no semantic analysis consistent with the real-world knowledge base can be derived. The difference is that with metaphor and metonymy, the meaning of the new usage is derivable from the meaning specified in the lexicon, given sufficiently rich referring functions and background knowledge. In contrast, the meaning of an idiom is idiosyncratic; i.e. the meaning must be stored in the lexicon, rather than built up compositionally from the meanings of its parts.[8] Metaphor and metonymy preserve use referring functions to map literal meaning to a new referent. Idioms do not generally use (non-idiomatic) literal meaning in any systematic way.

This does not mean, however, that idioms cannot be treated compositionally. As discussed in Gazdar, Klein, Pullum and Sag [Gazdar *et al.*, 1985a], semantic compositionality can be preserved for at least some idioms, according to the treatment of Wasow, Sag and Nunberg

---

[8] This is not to claim that the meaning of an idiom is not reconstructable, or that no compositionality is involved, but only that the semantic functions that determine the meaning of non-idiomatic usages are not sufficient for idioms.

[Wasow *et al.*, 1983]. For example, in the idiom "spill the beans," there is a separate lexical entry for the idiomatic form of the verb, with roughly the semantic content of divulge, and for the noun with the semantic content of roughly information. This approach does not require compositionality for all idioms. For example "kick the bucket" seems to lack any kind of compositionality; the entire phrase just means "die."

Learning a new idiom is thus equivalent to learning new senses for existing words, perhaps for several words of one string. This requires first, determining that the existing senses in the lexicon do not produce an acceptable interpretation, and second, that no acceptable interpretation can be produced from the application of any known referring functions to any existing senses. The latter assumes a complete set of basic referring functions, which is a topic of future work.

Having inferred that a sentence contains an idiom, the system should infer a new semantic sense for the relevant words, derived from context, while preserving their syntactic properties. By definition, idioms are invariably phrasal; *i.e.,* they involve more than one word. Many previous treatments of idioms have therefore stored them in the lexicon as multi-word lexical items, as in Zernik's work on the Phrasal Lexicon, [Zernik, 1988]. This approach is also available in the current system, as UNICORN is capable of representing multi-word lexical items.

However, there are drawbacks to this approach. Idioms vary greatly as to how frozen or flexible they are. Some are completely fixed, such as (8.18),[9] while others can allow verb inflection (8.19), or contentful modifiers (8.20).[10]

(8.18) We need to appeal for permission to the powers that be.

# When the time comes, we'll have to appeal for permission to the powers that will be.

(8.19) The old geezer finally kicked the bucket.

The old geezer is finally going to kick the bucket.

# The old geezer finally kicked the old oaken bucket.

---

[9] A hash-sign (#) in these sentences signifies that the sentence is grammatical, but cannot have the intended idiomatic interpretation.

[10] Even the most frozen idioms allow modifiers whose only function is to signal idiomaticity, such as (i).

(i) That's the way the proverbial cookie crumbles.

(8.20) The FBI kept annoyingly close tabs on John.

The compositional approach mentioned in [Gazdar *et al.*, 1985a] is therefore preferable, as it allows all of these degrees of flexibility to be presented in the lexical entries of the words making up the idiom. An HPSG-style lexicon has the power to represent this flexibility, while still maintaining the constraints that restrict the words of the idiom to co-occur only with each other.

This is possible because the SUBCAT list of a word makes reference to the entire SYNSEM feature of each of its elements, including the semantic content of those elements. Therefore, if there is a unique semantic entry for the relation denoted by each of the words in the idiom "spill the beans," then the SUBCAT list of the verb (whose semantic content contains the relation 'SPILL-DIVULGE') specifies that its object must be an entity whose semantic content contains the relation 'BEANS-INFORMATION.'[11]

There are many difficult problems here, which are still the subject of much ongoing research. I have not attempted to provide a complete solution for all the issues involved, but have sketched how a lexical acquisition module could interact productively with the analysis of idioms.

8.3 Storing the New Lexical Items Learned

Determining which newly learned information is to be stored in the lexicon of the system and which is to be discarded after use is not a trivial question. There are some cases in which it is advantageous to permanently store newly learned information and others in which it is not. For example, if the learning mechanism infers that a given sentence cannot be assigned an interpretation because a certain string of words is being used as an idiomatic expression, it will be to the system's advantage to store the idiomatic sense of the relevant words in the lexicon. This will enable the system to recognise the idiom in future encounters, without having to put forth the processing effort to infer its meaning again. This is the thrust of Uri Zernik's work on the Phrasal Lexicon. [Zernik, 1988]

---

[11] In [Pollard and Sag, 1987], the value of the SUBCAT feature is a list of signs; *i.e.*, complete feature structures of linguistic entities. This allows for the possibility of an element subcategorizing for a complement on the basis of the complement's phonology, among other things. This was seen as a weakness, as function words do not generally specify the phonology of their arguments. Therefore, in [Pollard and Sag, In press], the value of the SUBCAT feature was changed to a list of SYNSEM values of signs. However, for some frozen idioms, it may be desirable to specify that an element subcategorizes for another with a specified phonological form. This would be one way to explain the contrast of (ii).

(ii)a. The old geezer kicked the bucket.
b. # The old geezer kicked the pail

The alternative, adopted here, is to claim that while the literal meanings of "bucket" and "pail" are the same, "bucket" has an idiomatic sense which is not shared by "pail."

However, when the system infers the referent of a proper noun, such as determining the person in the world who is referred to by the name "John," this is probably not information that should be stored permanently. That is, the system should not assume that this name will refer to the same person when it is encountered in unrelated contexts in subsequent texts. Therefore the referent of the name should be inferred, but not stored for longer than the length of the current text.

On the other hand, there are proper names, such as "Churchill", whose referent should be learned and permanently stored, either as a part of global knowledge or relative to a particular domain of real-world knowledge. The decision of whether to store the referent of a given name must therefore be made in conjunction with a theory of discourse representation, part of which includes pointers to the domain of the real world being discussed. As it is currently implemented, the system handles unknown proper names in the same way as any other unknown lexical items, but it may be the case that greater efficiency can be obtained by having special procedures test for names of persons or other entities, as in [Jacobs and Rau, 1988].

The same issue arises in the case of a word of one syntactic category being used as another. Here the question is whether the derived category should be added to the lexicon. It requires some computational effort to determine a new category for a known word. But there is also some expense associated with storing it. Therefore, if the system encounters a novel category for a known word only once, it should just compute it and move on. However, if the novel category is encountered frequently, it should not be recomputed each time. Rather, the system should notice that it occurs frequently, and store it.

For example, take the denominal verb "impact," meaning 'have an effect on.'[12] The first time this is encountered, it will be analysed and its derived meaning inferred. At some point, though, the system should add the verbal form to the lexicon, as a non-derived lexical item. However, there is also an expense in keeping track of how many times a word is used in a given derived category. The trade-off in efficiency between computing derived categories and storing them is an empirical issue, which should be explored in future work.

---

[12] There is a non-derived verbal sense of "impact," meaning "to pack or wedge," but this is rarely used outside of dentists' offices, and there only in the past participial form.

8.4 Generation and Incomplete Specification

The work on lexical acquisition described in this dissertation has been discussed in the context of language understanding, as is common with research on language learning. In general, systems acquire language in order to understand it more completely. But just as a person learns language not only to understand the utterances of others, but also to communicate with them, so also a Natural Language system that performs both understanding and generation should learn new lexical items not only to analyse utterances but to produce them as well.

One difference between this system and others for language acquisition is that it allows the partial learning of new lexical items. That is, viewing a lexical item as a set of slot-filler pairs, a lexical entry may be well-formed even with some of its slot values unspecified. This allows the system to work with a word while having only a partial representation of it. For example, we may know (i) nothing about a word except that it is a verb, (ii) nothing about a word except that it is a verb referring to an action performable by a person or (iii) that the action is performable by a person on an inanimate object, etc.

The ability to work with partial representations of lexical items allows people to use words, often correctly, without completely understanding them. For example, if John utters sentence (8.21), the hearer may not know what the verb "glarf" means, but can reasonably infer that it is something that a person can do to a carburetor.

(8.21) I'm going to go glarf my carburetor now.

The hearer does not know what John is doing, in the sense of being able to describe his activity by means of a paraphrase, or break it down into subactions. But if Mary asks what John is doing, the hearer of (8.21) can answer that he is glarfing his carburetor, which may well be sufficient for Mary. Thus, the hearer can use the verb "glarf" in generation before completely understanding it.

The learning system described here, acquiring only that information which is available from context, should similarly be able to deal with partial information. In learning the word "glarf," one possibility, as a lower bound of a lexical version space, is that the action denoted by "glarf" is only something that a person can do to a carburetor. In fact, after only this single encounter with the new word, the system will construct one possibility in which the action is something that only **John** can do to a carburetor. More general possibilities are maintained higher in the lexical version space.

In order to be useful in the context of a query-answering system, the language learning mechanism must be able to deal with partial knowledge. If only the selectional restrictions of a new word can be inferred from the input, the system should respond to the utterance using this information. In the case of (8.22), the system will not be able to answer "yes" or "no" without figuring out the meaning of "glarf." Nevertheless, it should recognize (8.22) as a request for information as to the proper action to take. From this knowledge, it will be able to give an answer such as (8.23)a or (8.23)b.

(8.22) Should I glarf on this street?

(8.23) a. Turn left on this street.
b. Go straight past this street and keep following the road you're on.

Attempting to use a word in generation without completely understanding it may in some cases lead to mistakes. This is certainly a drawback, assuming that the goal is to create a system that performs without error. However, it undoubtedly reflects the way people go about learning. Dulay and Burt [Dulay and Burt, 1974] describe a number of language errors, and argue that they are an essential step in language learning.

In order to utilize the power of language learning in generation while minimizing the risk of errors, the best approach would be to prefer words which are completely known over those with only partial specifications, but to use partially learned forms where no good alternative exists. This is the approach taken in Zernik's RINA system [Zernik, 1988]. In generation, this may result in sentences that are ungrammatical, but communicate the idea effectively, much like a non-native speaker of English.

In generating a natural language utterance to communicate a given (machine-internal) concept, the system would choose an understood word over one not completely understood, if both are available. This means that there may be lexical items which can be processed in understanding that will never show up in generation. Again, this parallels the case with people.

It has long been an axiom among developmental psycholinguists that comprehension precedes production. This tendency will now show up as well in an NLP system which acquires new words.

Chapter 9

Conclusions

This chapter summarises the learning mechanism described in this dissertation, together with the foundational work within which it operates and the implications that it has for other work.

Chapter Two provided a comparison of this work with other approaches to lexical acquisition, along two dimensions. Regarding the dimension of the starting point of the system, it was shown that systems that claim to build a lexicon from scratch are not in principle different from those that extend the lexicon of a system which already has language understanding capabilities. This is because, in both cases, the system must use information regarding possible lexical categories of a language. I have therefore chosen to make this information explicit, and speak in terms of extending a lexicon which is already complete, in a well-defined sense. That is, the system has a complete inventory of the syntactic categories of the language and relevant type information about each category.

Along the dimension of the method of learning, I have chosen to explore the option of learning solely through context, both linguistic and non-linguistic. Various difficulties were seen with the other major approaches which have been advocated in the literature. Learning by being told is both burdensome to the user and highly susceptible to error, in that it requires eliciting knowledge from native speakers, who for the most part are unable to articulate the principles that they use unconsciously in producing linguistic utterances. Learning from statistics can produce initially promising results, but has inherent limitations, as the underlying principles of language structure and use are ignored. The application of existing general models of learning suffers from the drawbacks of those models, primarily in generating an unmanageably large number of hypothesized lexicons.

Chapter Three provides the theoretical background of the framework within which the learning mechanism is couched. This chapter also motivates the choices of the particular grammatical framework and parsing strategies employed by the learning mechanism. Unification-Based Grammar in general, and Head-Driven Phrase Structure Grammar in particular, was shown to have the advantage of making predictions as to the feature structure of an unknown, which are further instantiated in subsequent parsing. Gerdemann's UNICORN [Gerdemann and Hinrichs, 1988], [Gerdemann, 1991], implementing Earley's algorithm, was shown to be an efficient means of taking advantage of such a framework in lexical acquisition.

A lexical acquisition system cannot operate in a vacuum, but only in conjunction with a grammar and a database of real-world knowledge. These components are described in Chapter Four. In general, the grammar is a straightforward implementation of the version of HPSG set forth in [Pollard and Sag, In press]. However, certain differences from that version of the theory, as well as additions to it, were found to be necessary in order to achieve syntactic coverage of the given data. The most drastic of these modifications is a reformulation of the feature structure of the semantic content of verbs. This modification, motivated by efficiency concerns in encoding of the semantics of adverbs and connecting the parser with the database of real-world knowledge, is described in Chapter Five.

The algorithm for inferring the syntax and semantics of new lexical items is described in Chapters Six and Seven. Chapter Six tells how, on a single encounter with an unknown word, the system constructs a space of hypotheses about its feature structure. The hypothesis space is encoded in a data structure called a **lexical version multi-space**, consisting of a disjunctive set of **lexical version spaces**. Each lexical version space represents a distinct prediction made for the feature structure of the unknown word.

A lexical version space consists of a single upper bound and a set of lower bounds. An upper bound is established by the predictions from by Earley's algorithm, using a set of features specified by the grammar writer in the restrictor, which delimits the feature paths referenced by the scanner. In addition, the upper bound includes information available about the particular syntactic category, as given in the lexical type hierarchy of HPSG.

Among other things, the type hierarchy specifies the formal semantic properties associated with a particular syntactic category. This yields a structure within which to place a variable representing the association of the lexical item within a particular concept of the real-world knowledge base. This variable is instantiated in the process of semantic checking, as a disjunction of concepts which satisfy the constraints imposed by other elements of the sentence.

What makes lexical acquisition solely from context feasible is the ability to combine feature structure information about the new word obtained from multiple encounters with it. Chapter Seven describes the procedure for combining the hypotheses generated from a single encounter with an unknown with the hypothesis set constructed from previous encounters with that item, to form an updated hypothesis set.

The first step in the updating procedure is to combine the sets of upper bounds. Here, every element of one set of upper bounds is unified with each element of the other set, an operation called **cross-unification.** In theory, this leads to a potentially larger set of upper bounds in the output. However, because the elements of each input set generally are mutually non-unifiable, cross-unification in practice has the effect of intersection.

For each upper bound in the updated lexical version multi-space, the system then combines the corresponding sets of lower bounds. This involves finding the feature structure information which they share; *i.e.*, the maximal amount of information which satisfies both the constraints of the current input data and those constraints imposed by previous input, on a given hypothesis for the category of the new word. This again employs a pair-wise comparison of the two sets of lower bounds. The results of these comparisons are checked for uninformativeness, duplication, and subsumption of existing lower bounds, and those which remain form the set of lower bounds corresponding to the given upper bound in the updated lexical version multi-space.

Chapter Eight discusses a number of issues for further research raised by the work of this dissertation. Sketches of a treatment for these issues are given, although they have not been implemented in the current system. Several of these issues relate to the HPSG concept of lexical rules. Such rules can be use in an efficient representation of the lexicon, by relating the various morphological forms of a word.

Lexical rules also provide a promising means of implementing **referring functions**, which in turn would allow the system to analyze a variety of metonymic relations. These include both conventional and *ad hoc* associations between an entity mentioned in an utterance, and the entity to which the speaker intends to refer. Similar lexical rules offer a possible treatment of metaphor, and of the phenomenon of a word of one syntactic category productively used in a different category.

The acquisition of new lexical items is both a central problem of language learning, and a complex issue, involving the interaction of syntax, semantics, and extra-linguistic knowledge and heuristics. This work has shown that the various facets of such learning can be accomplished from context, without placing an inordinate burden on the user, through the means of unification-based grammar formalisms, together with an interleaved domain-specific real-world knowledge base.

Appendix A

Sample Grammar

The following grammar contains all of the rules used in lexical acquisition. However, it does not contain all the lexical items of that grammar, but only a few representatives of each lexical category.

```
%% Name of grammar: hpsg

%% This grammar should be exactly like Pollard and Sag, Vol. 2, with
%% the following exceptions:

%% 1.  The obliqueness ordering is subject-last, as in Vol. 1, rather
%%     than subject-first, as in Vol. 2.

%% 2.  The semantics of the arguments of verbs is of type <content>,
%%     rather than <content index>.  Otherwise, the restrictions on
%%     a noun's semantics are not reflected on the semantics of the
%%     verb.

%% 3.  The treatment of quantification is rudimentary.  Determiners
%%     have enough info to let them be parsed, but not much more.

%% 4.  The treatment of relative clauses is far short of what is
%%     given in Vol. 2.  The grammar handles only subject relatives,
%%     and has no traces.

%% 5.  The top-level feature <maj> is a necessary artifact of unicorn.
%%     Future work should implement an explicit type system, so I can
%%     say that, for example, a transitive verbs takes as its first
%%     complement something of type noun.  As a compromise, I've
%%      added a head feature not found in Vol. 2, called maj, and made
%%      it always re-entrant with the top-level feature maj.

%%  6.  I have encoded bar-levels explicitly in the grammar, and done
%%      away with the <lex> feature.  This is in accordance with
%%      Chapter 9 of Volume 2, though different from the rest of the
%%      book.

%%  7.  For the sake of the real-world knowledge component, it's
%%      crucial that adjectives add their restrictions onto the end
%%      of the restr list rather than the beginning.  This is because
%%      we need to have access to the entity_type, which is found in
%%      the restriction associated with the N itself.
```

A.1 General Principles of the Grammar

```
%%  The Head Feature Principle:
%%  The HEAD value of any headed phrase is structure-shared with
%%  the HEAD value of the head daughter.
%%      The first two lines below have nothing to do with the HFP,
%%      but this is a  generalization that applies to all headed
%%      structures, so it's convenient to put it here.

define hfp:
<x0 synsem local category head maj> = <x0 maj>
<x0 maj> = <x0 dtrs headdtr maj>
<x0 synsem local category head> = <x0 dtrs headdtr synsem local category head>.


%%  The SPEC Principle:
%%  If a non-head daughter in a headed structure bears a SPEC value,
%%  it is token-identical to the SYNSEM value of the head daughter.
%%      Since we can't say "non-head," we need different versions for
%%      spr daughters and marker daughters.

define spec-spr:
<x0 dtrs sprdtrs first synsem local category head spec synsem> =
    <x0 dtrs headdtr synsem>.

define spec-mkr:
<x0 dtrs mkrdtr synsem local category head spec synsem> =
    <x0 dtrs headdtr synsem>.
```

```
%% Preliminary version of the context principle:
%% In any structure with two daughters, the context of the mother
%% is the union of those of the daughters.  This is known to be too
%% strong, as described in Chapter Eight, but capturing the way
%% context actually works necessitates reference to extra-linguistic
%% information.

define context2:
<x0 synsem local context list> = <x1 synsem local context list>
<x1 synsem local context eol> = <x2 synsem local context list>
<x0 synsem local context eol> = <x2 synsem local context eol>.

define context3:
<x0 synsem local context list> = <x1 synsem local context list>
<x1 synsem local context eol> = <x2 synsem local context list>
<x2 synsem local context eol> = <x3 synsem local context list>
<x0 synsem local context eol> = <x3 synsem local context eol>.


%% Preliminary version of the Quantifier Inheritance Principle:

define qip2:
<x0 qstore list> = <x1 qstore list>
<x1 qstore eol> = <x2 qstore list>
<x0 qstore eol> = <x2 qstore eol>.

define qip3:
<x0 qstore list> = <x1 qstore list>
<x1 qstore eol> = <x2 qstore list>
<x2 qstore eol> = <x3 qstore list>
<x0 qstore eol> = <x3 qstore eol>.


%%  The following phon rules are for encoding surface word order.
%%  They do NOT give the simplest possible representation (which would
%%  be just concatenation, putting things in the correct order -- as
%%  is done in the default rules at the end), but aim to be as principled
%%  as possible.

%%  In a head-subj structure, the head goes last.

define phonHeadSubj:
<x0 phon list> = <x0 dtrs subjdtrs first phon list>
```

```
<x0 dtrs subjdtrs first phon eol> = <x0 dtrs headdtr phon list>
<x0 phon eol> = <x0 dtrs headdtr phon eol>.


%%  In a head-comp structure, the lexical head goes first.
%%  This version is for heads with exactly one comp dtr.

define phonHeadComp1:
<x0 phon list> = <x0 dtrs headdtr phon list>
<x0 dtrs headdtr phon eol> = <x0 dtrs compdtrs first phon list>
<x0 phon eol> = <x0 dtrs compdtrs first phon eol>.

%%  This version is for head with exactly two comp dtrs.

define phonHeadComp2:
<x0 phon list> = <x0 dtrs headdtr phon list>
<x0 dtrs headdtr phon eol> = <x0 dtrs compdtrs first phon list>
<x0 dtrs compdtrs first phon eol> = <x0 dtrs compdtrs rest first phon list>
<x0 phon eol> = <x0 dtrs compdtrs rest first phon eol>.


%%  In a head-spr structure, a non-V head goes last.

define phonNonVHeadSpr:
<x0 phon list> = <x0 dtrs sprdtrs first phon list>
<x0 dtrs sprdtrs first phon eol> = <x0 dtrs headdtr phon list>
<x0 phon eol> = <x0 dtrs headdtr phon eol>.

%%  In a head-spr structure, a V head goes first.

define phonVHeadSpr:
<x0 phon list> = <x0 dtrs headdtr phon list>
<x0 dtrs headdtr phon eol> = <x0 dtrs sprdtrs first phon list>
<x0 phon eol> = <x0 dtrs sprdtrs first phon eol>.


%%  In a head-mkr structure, the head goes last.

define phonHeadMkr:
<x0 phon list> = <x0 dtrs mkrdtr phon list>
<x0 dtrs mkrdtr phon eol> = <x0 dtrs headdtr phon list>
<x0 phon eol> = <x0 dtrs headdtr phon eol>.
```

```
%% In a head-adjunct structure with a non-V head and a lexical
%% adjunct, the adjunct goes first.

define phonHeadLexAdj:
<x0 phon list> = <x0 dtrs adjdtr phon list>
<x0 dtrs adjdtr phon eol> = <x0 dtrs headdtr phon list>
<x0 phon eol> = <x0 dtrs headdtr phon eol>.

%% In a head-adjunct structure with a non-V head and a phrasal
%% adjunct, the adjunct goes last.

define phonHeadPhrAdj:
<x0 phon list> = <x0 dtrs headdtr phon list>
<x0 dtrs headdtr phon eol> = <x0 dtrs adjdtr phon list>
<x0 phon eol> = <x0 dtrs adjdtr phon eol>.


%% Default phon rule for a structure with two daughters:

define phon2:
<x0 phon list> = <x1 phon list>
<x1 phon eol> = <x2 phon list>
<x0 phon eol> = <x2 phon eol>.
```

A.2 Rule Schemata

A.2.1 Schemata for Headed Structures

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Subject-Head Schema                        %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

define subj-head:
phonHeadSubj
hfp
context2
qip2
<x0 synsem local category subj> = end
<x0 synsem local category spr> = end
<x0 synsem local category comps> = end
<x0 synsem local category bar> = two
<x0 synsem local conj> =  nil
```

```
<x0 synsem local content> = <x0 dtrs headdtr synsem local content>
<x0 dtrs headdtr synsem local category spr> = end
<x0 dtrs headdtr synsem local category comps> = end
<x0 dtrs headdtr synsem local category subj first synsem> =
    <x0 dtrs subjdtrs first synsem>
<x0 dtrs headdtr synsem local category bar> = two
<x0 dtrs headdtr synsem local conj> = nil
<x0 dtrs subjdtrs rest> = end.

%%  Declarative sentences
%%  In X-bar notation written as: s -> n2 v2

v -> n v:
subj-head
<x0 dtrs subjdtrs first> = <x1>
<x0 dtrs headdtr> = <x2>.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Specifier-Head Schema                       %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

define spec-head-zero:
hfp
<x0 phon> = <x0 dtrs headdtr phon>
<x0 synsem local category comps> = end
<x0 synsem local category spr> = end
<x0 synsem local category subj> = <x0 dtrs headdtr synsem local category subj>
<x0 synsem local category bar> = two
<x0 synsem local conj> =  nil
<x1 synsem local conj> =  nil
<x0 synsem local content> = <x0 dtrs headdtr synsem local content>
<x0 synsem local context> = <x0 dtrs headdtr synsem local context>
<x0 qstore> = <x0 dtrs headdtr qstore>
<x0 dtrs headdtr synsem local category comps> = end
<x0 dtrs headdtr synsem local category spr> = end
<x0 dtrs headdtr synsem local category bar> = one.

x0 -> x1:
spec-head-zero
<x0 dtrs headdtr> = <x1>.


define specifier-head:
```

```
hfp
spec-spr
context2
qip2
<x0 synsem local category comps> = end
<x0 synsem local category subj> =
    <x0 dtrs headdtr synsem local category subj>
<x0 synsem local category spr> = end
<x0 synsem local category bar> = two
<x0 synsem local category marking> =
    <x0 dtrs headdtr synsem local category marking>
<x0 synsem local conj> =  nil
<x2 synsem local conj> =  nil
<x0 synsem local content> = <x0 dtrs headdtr synsem local content>
<x0 dtrs headdtr synsem local category comps> = end
<x0 dtrs headdtr synsem local category bar> = one
<x0 dtrs headdtr synsem local category spr first synsem> =
    <x0 dtrs sprdtrs first synsem>
<x0 dtrs sprdtrs rest> = end.

%%  noun phrase with a determiner
%%  n2 -> det n1

n -> det n:
phonNonVHeadSpr
specifier-head
<x0 dtrs sprdtrs first> = <x1>
<x0 dtrs headdtr> = <x2>.

%% back up a little
%%         ========
v -> v adv:
phonVHeadSpr
specifier-head
<x0 dtrs sprdtrs first> = <x2>
<x0 dtrs sprdtrs first synsem local category head mod> =  nil
<x0 dtrs sprdtrs first synsem local category bar> = two
<x0 dtrs headdtr> = <x1>
<x1 synsem local category head form> = {base finite prog}.

%% six miles away from here
%% =========
adj -> adv adj:
phonNonVHeadSpr
```

```
specifier-head
<x0 dtrs sprdtrs first> = <x1>
<x0 dtrs sprdtrs first synsem local category head mod> =  nil
<x0 dtrs sprdtrs first synsem local category bar> = two
<x0 dtrs headdtr> = <x2>.

%% approximately three blocks down the road
%% ==========================
adv -> adv adv:
phonNonVHeadSpr
specifier-head
<x0 dtrs sprdtrs first> = <x1>
<x0 dtrs sprdtrs first synsem local category head mod> =  nil
<x0 dtrs sprdtrs first synsem local category bar> = two
<x0 dtrs headdtr> = <x2>.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Complement-Head Schemata                   %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% This version of the Complement-Head Schema works only for cases
%% where there are zero complement daughters.

define comp-head-zero:
hfp
<x0 phon> = <x0 dtrs headdtr phon>
<x0 dtrs headdtr synsem local category bar> = zero
<x0 dtrs headdtr synsem local category comps> = end
<x0 synsem local category subj> =
    <x0 dtrs headdtr synsem local category subj>
<x0 synsem local category comps> =
    <x0 dtrs headdtr synsem local category comps>
<x0 synsem local category spr> =
    <x0 dtrs headdtr synsem local category spr>
<x0 synsem local category marking> =
    <x0 dtrs headdtr synsem local category marking>
<x0 synsem local category bar> = one
<x0 synsem local conj> =  nil
<x0 dtrs headdtr synsem local conj> = nil
<x0 synsem local content> = <x0 dtrs headdtr synsem local content>
<x0 synsem local context> = <x0 dtrs headdtr synsem local context>
<x0 qstore> = <x0 dtrs headdtr qstore>.
```

```
x0 -> x1:
comp-head-zero
<x0 dtrs headdtr> = <x1>.


%% This version of the Complement-Head Schema works only for cases
%% where there is exactly one complement daughter.  It includes
%%  A lexical noun with a complement.
%%  A verb phrase with expands to a lexical verb plus whatever it
%%  subcategorizes for, including all of the following:
%%  1.  Modal auxiliaries.
%%  2.  Verbs that take an infinitival vp as complement, such as
%%          want to turn right
%%  3.  Verbs that take a progressive vp as complement, such as
%%          keep going straight
%%  4.  Infinitival TO plus a v2 in base form,
%%          to turn right
%%  5.  Verb that subcategorizes for an obligatory adv2, such as HEAD
%%          head toward flb
%%  6.  Transitive verb, subcategorizes for a direct object.
%%          v1 -> v n2
%%  Adjectival prepositional phrases
%%     p1 -> p n2
%%  Adjectives that take obligatory complements
%%     unable to turn left
%%  Adverbial prepositional phrases
%%     p1 -> p n2

define comp-head-one:
hfp
context2
qip2
phonHeadComp1
<x0 synsem local category subj> = <x0 dtrs headdtr synsem local category subj>
<x0 synsem local category comps> = end
<x0 synsem local category bar> = one
<x0 synsem local conj> = nil
<x0 synsem local content> = <x0 dtrs headdtr synsem local content>
<x0 dtrs compdtrs rest> = end
<x0 dtrs headdtr synsem local conj> = nil
<x0 dtrs headdtr synsem local category comps first synsem> = <x0 dtrs compdtrs first synsem
<x0 dtrs headdtr synsem local category comps rest> = end
<x0 dtrs headdtr synsem local category bar> = zero.
```

```
x0 -> x1 x2:
comp-head-one
<x0 dtrs headdtr> = <x1>
<x0 dtrs compdtrs first> = <x2>.


%% This version of the complement-head schema works only for cases
%% where there are exactly two complement daughters.

define comp-head-two:
hfp
phonHeadComp2
context3
qip3
<x0 synsem local category comps> =
    <x0 dtrs headdtr synsem local category comps rest rest>
<x0 synsem local category comps> = end
<x0 synsem local category bar> = one
<x0 synsem local conj> = <x0 dtrs headdtr synsem local conj>
<x0 synsem local content> = <x0 dtrs headdtr synsem local content>
<x0 dtrs headdtr synsem local category bar> = zero
<x0 dtrs headdtr synsem local category comps first synsem> =
    <x0 dtrs compdtrs first synsem>
<x0 dtrs headdtr synsem local category comps rest first synsem> =
    <x0 dtrs compdtrs rest first synsem>
<x0 dtrs compdtrs rest rest> = end.

x0 -> x1 x2 x3:
comp-head-two
<x0 dtrs headdtr> = <x1>
<x0 dtrs compdtrs first> = <x2>
<x0 dtrs compdtrs rest first> = <x3>.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Marker-head Schema                         %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% To avoid proliferation of categories, I'm calling
%% case-marking prepositions comps.

define marker-head:
hfp
spec-mkr
```

```
context2
phonHeadMkr
qip2
<x0 synsem local category comps> = end
<x0 synsem local category spr> = end
<x0 synsem local category subj> = end
<x0 synsem local category bar> = two
<x0 synsem local category marking> =
    <x0 dtrs mkrdtr synsem local category marking>
<x0 synsem local conj> = <x0 dtrs headdtr synsem local conj>
<x0 synsem local content> = <x0 dtrs headdtr synsem local content>
<x0 dtrs headdtr synsem local category comps> = end
<x0 dtrs headdtr synsem local category spr> = end
<x0 dtrs headdtr synsem local category subj> = end
<x0 dtrs headdtr synsem local category bar> = two.

x0 -> comp x2:
marker-head
<x0 dtrs mkrdtr> = <x1>
<x0 dtrs headdtr> = <x2>.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Adjunct-Head Schema                        %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

define adjunct-head:
hfp
context2
qip2
<x0 dtrs adjdtr synsem local category head mod synsem> =
    <x0 dtrs headdtr synsem>
<x0 dtrs adjdtr synsem local category comps> = end
<x0 dtrs adjdtr synsem local category spr> = end
<x0 dtrs adjdtr synsem local category bar> = two
<x0 dtrs adjdtr synsem local conj> = nil
<x0 synsem local content> = <x0 dtrs adjdtr synsem local content>
<x0 synsem local conj> = <x0 dtrs headdtr synsem local conj>
<x0 synsem local category subj> = <x0 dtrs headdtr synsem local category subj>
<x0 synsem local category comps> =
    <x0 dtrs headdtr synsem local category comps>
<x0 synsem local category spr> = <x0 dtrs headdtr synsem local category spr>
<x0 synsem local category bar> = <x0 dtrs headdtr synsem local category bar>
<x0 synsem local category marking> =
    <x0 dtrs headdtr synsem local category marking>.


%% n1 -> adj n
%%  e.g.   third right

n -> adj n:
adjunct-head
phon2
<x0 dtrs adjdtr> = <x1>
<x0 dtrs headdtr> = <x2>
<x1 synsem local category subj> = end
<x2 synsem local category bar> = one.


%%  n1 -> n adj
%%  e.g. stoplight on the corner

n -> n adj:
adjunct-head
phon2
<x0 dtrs headdtr> = <x1>
<x0 dtrs adjdtr> = <x2>
<x1 synsem local category bar> = one.


%%  Pre-sentential adverbs.
%%     "On elm, take a right"
%%  Could these be filler-gap structures, and actually VP-modifying adverbs?

v -> adv v:
adjunct-head
phon2
<x0 dtrs adjdtr> = <x1>
<x0 dtrs headdtr> = <x2>
<x2 synsem local category subj> = end.


%%  Optional vp-modifier.
%%     "Take Green north until you get to flb".
%%     v2 -> v2 adv2

v -> v adv:
adjunct-head
```

```
phon2
<x0 dtrs headdtr> = <x1>
<x0 dtrs adjdtr> = <x2>
<x1 synsem local category subj rest> = end
<x1 synsem local category head form> = {base finite prog}.

%%  Optional verb modifier
%%      "John quickly turned the corner."
v -> adv v:
adjunct-head
phon2
<x0 dtrs adjdtr> = <x1>
<x0 dtrs headdtr> = <x2>
<x2 synsem local category subj rest> = end.
```

A.2.2 Schema for Non-Headed Structures:

```
%%  Coordination:
%%  Note:  These rules are computationally expensive.

%% Binary Coordination: both a and b.

define binarycoord:
phon2
context2
qip2
<x0 dtrs conjunct1> = <x1>
<x0 dtrs conjunct2> = <x2>
<x0 synsem local category> = <x1 synsem local category>
<x0 synsem local category> = <x2 synsem local category>
<x0 synsem local category head maj> = <x0 maj>
<x0 synsem local conj> = nil
<x1 synsem local conj> = both
<x2 synsem local conj> = and.

n -> n n:
binarycoord
<x0 synsem local content index num> = plural
<x0 synsem local content restr first relation conj1> =
    <x1 synsem local content>
<x0 synsem local content restr first relation conj2> =
    <x2 synsem local content>
<x0 synsem local content restr first instance> =
    <x0 synsem local content index>
<x0 synsem local content restr rest> = end.

v -> v v:
binarycoord
<x0 synsem local content conj1> = <x1 synsem local content>
<x0 synsem local content conj2> = <x2 synsem local content>.

adj -> adj adj:
binarycoord
<x0 synsem local content index> = <x1 synsem local content>
<x0 synsem local content index> = <x2 synsem local content>
<x0 synsem local content restr first relation conj1> =
    <x1 synsem local content restr>
<x0 synsem local content restr first relation conj2> =
    <x2 synsem local content restr>
<x0 synsem local content restr first instance> =
    <x0 synsem local content index>
<x0 synsem local content restr rest> = end.

adv -> adv adv:
binarycoord
<x0 synsem local content psoa> = <x1 synsem local content psoa>
<x0 synsem local content psoa> = <x2 synsem local content psoa>
<x0 synsem local content restr first relation conj1> =
    <x1 synsem local content restr>
<x0 synsem local content restr first relation conj2> =
    <x2 synsem local content restr>
<x0 synsem local content restr first instance> =
    <x0 synsem local content index>
<x0 synsem local content restr rest> = end.


%% Iterative Coordination
%% Of the following two rules,
%% a, b, c and d       - uses the first at the top, and repeated tokens
%%                       of the second in the middle.
%% a and b and c and d - uses the first at the top, and alternates
%%                       between the first and the head-marker rule,
%%                       with the latter at the bottom.

n -> n n:
phon2
context2
```

```
qip2
<x0 dtrs conjunct1> = <x1>
<x0 dtrs conjunct2> = <x2>
<x0 synsem local category> = <x1 synsem local category>
<x0 synsem local category> = <x2 synsem local category>
<x0 synsem local category head maj> = <x0 maj>
<x0 synsem local conj> = nil
<x1 synsem local conj> = nil
<x2 synsem local conj> = and
<x0 synsem local content index num> = plural
<x0 synsem local content restr first relation conj1> =
    <x1 synsem local content>
<x0 synsem local content restr first relation conj2> =
    <x2 synsem local content>
<x0 synsem local content restr first instance> =
    <x0 synsem local content index>
<x0 synsem local content restr rest> = end.

n -> n n:
phon2
context2
qip2
<x0 dtrs conjunct1> = <x1>
<x0 dtrs conjunct2> = <x2>
<x0 synsem local category> = <x1 synsem local category>
<x0 synsem local category> = <x2 synsem local category>
<x0 synsem local category head maj> = <x0 maj>
<x0 synsem local conj> = and
<x1 synsem local conj> = nil
<x2 synsem local conj> = and
<x0 synsem local content index num> = plural
<x0 synsem local content restr first relation conj1> =
    <x1 synsem local content>
<x0 synsem local content restr first relation conj2> =
    <x2 synsem local content>
<x0 synsem local content restr first instance> =
    <x0 synsem local content index>
<x0 synsem local content restr rest> = end.


%%  N-N appositives

n -> n n:
hfp
```

```
context2
phon2
qip2
<x0 dtrs headdtr> = <x2>
<x0 dtrs adjdtr> = <x1>
<x0 synsem local category spr> = <x0 dtrs headdtr synsem local category spr>
<x1 synsem local category> = <x2 synsem local category>
<x0 dtrs headdtr synsem local category spr> = end
<x0 dtrs adjdtr synsem local category spr> = end
<x0 synsem local category bar> = two
<x1 synsem local category bar> = two
<x2 synsem local category bar> = two
<x0 synsem local category marking> = <x1 synsem local category marking>
<x0 synsem local category marking> = <x2 synsem local category marking>
<x0 synsem local conj> = nil
<x1 synsem local conj> = nil
<x2 synsem local conj> = nil
<x0 synsem local content index> = <x0 dtrs headdtr synsem local content index>
<x0 synsem local content restr first> =
    <x0 dtrs headdtr synsem local content restr>
<x0 synsem local content restr rest first relation> = associated
<x0 synsem local content restr rest first associator> =
    <x0 dtrs adjdtr synsem local content index>
<x0 synsem local content restr rest first associated> =
    <x0 synsem local content index>
<x0 synsem local content restr rest rest> =
    <x0 dtrs adjdtr synsem local content restr>.
```

A.3 Lexical Rules

```
define lexrule:
<x0 phon> = <x0 dtrs unarydtr phon>
<x0 synsem local category bar> = zero
<x0 synsem local conj> = <x0 dtrs unarydtr synsem local conj>
<x0 synsem local context> = <x0 dtrs unarydtr synsem local context>
<x0 dtrs unarydtr synsem local category bar> = zero
<x0 qstore> = <x0 dtrs unarydtr qstore>.

%% The following defines are parallel to those given for
%% syntactic categories in the lexicon below, but are
%% used to specify the left-hand side of a category-changing
%% lexical rule.  Right now, there are so few lexical rules
%% that this is not a savings in space, but it will make it
%% eaiser to write new lexical rules.

define lhs-subst:
<x0 synsem local category head maj> = <x0 maj>
<x0 synsem local conj> = nil.


define lhs-adverb-mod-vp:
lhs-subst
<x0 synsem local category head maj> = adv
<x0 synsem local category head mod synsem local category head maj> = v
<x0 synsem local category head mod synsem local category head form> =
    {base finite ppart prog}
<x0 synsem local category head mod synsem local category subj first
    synsem local category head maj> = n
<x0 synsem local category head mod synsem local category subj rest> = end
<x0 synsem local category head mod synsem local category comps> = end
<x0 synsem local category head mod synsem local category bar> = two
<x0 synsem local content soa-arg> =
    <x0 synsem local category head mod synsem local content>.


%%    John came home early to surprise his wife.
%%                         =====================
adv -> v:
lexrule
lhs-adverb-mod-vp
<x0 dtrs unarydtr> = <x1>
<x1 synsem local category head form> = infinitive
<x0 synsem local category head mod synsem local category subj> =
    <x1 synsem local category subj>
<x0 synsem local category comps> = <x1 synsem local category comps>
<x0 synsem local content relation> = purpose
<x0 synsem local content goal> = <x1 synsem local content>.

%%    He pulled out, making a left turn
%%                     ==================
adv -> v:
lexrule
lhs-adverb-mod-vp
<x0 dtrs unarydtr> = <x1>
<x1 synsem local category head form> = prog
<x0 synsem local category head mod synsem local category subj> =
    <x1 synsem local category subj>
```

BEST COPY AVAILABLE

```
<x0 synsem local category comps> = <x1 synsem local category comps>
<x0 synsem local content relation> = <x1 synsem local content>.


define lhs-commonnoun:
lhs-subst
<x0 synsem local category head maj> = n
<x0 synsem local category head form> = norm
<x0 synsem local category head case> = {acc nom}
<x0 synsem local category marking> = unmarked
<x0 synsem local content index gend> = neuter
<x0 synsem local content index per> = third
<x0 synsem local content restr first instance> =
    <x0 synsem local content index>
<x0 synsem local content restr rest> = end.

%%   Turning left would be a mistake
%%   ============
n -> v:
lexrule
lhs-commonnoun
<x0 dtrs unarydtr> = <x1>
<x1 synsem local category head form> = prog
<x0 synsem local category comps> = <x1 synsem local category comps>
<x1 synsem local category subj rest> = end
<x0 synsem local content> = <x1 synsem local content>.


define lhs-adjective:
lhs-subst
<x0 synsem local category head maj> = adj
<x0 synsem local category head mod synsem local category head maj> = n
<x0 synsem local content index> =
    <x0 synsem local category head mod synsem local content index>
<x0 synsem local content restr first> =
    <x0 synsem local category head mod synsem local content restr first>
<x0 synsem local content restr rest rest> =
    <x0 synsem local category head mod synsem local content restr rest>.

define lhs-modifying-adj:
lhs-adjective
<x0 synsem local category subj> = end.

define lhs-predicative-adj:
```

```
lhs-adjective
<x0 synsem local category subj first synsem> =
    <x0 synsem local category head mod synsem>.

%%    A pick-up turning left cut me off.
%%              ============
adj -> v:
lexrule
lhs-adjective
<x0 dtrs unarydtr> = <x1>
<x1 synsem local category head form> = prog
<x0 synsem local category head mod synsem local content> =
    <x1 synsem local category subj first synsem local content>
<x0 synsem local category comps> = <x1 synsem local category comps>
<x1 synsem local category subj rest> = end
<x0 synsem local content restr rest first> = <x1 synsem local content>.

%%    A pick-up, dented on the left side, cut me off.
%%               =======================
adj -> v:
lexrule
lhs-adjective
<x0 dtrs unarydtr> = <x1>
<x1 synsem local category head form> = ppart
<x0 synsem local category head mod synsem local content> =
    <x1 synsem local category comps first synsem local content>
<x0 synsem local category comps> = <x1 synsem local category comps rest>
<x1 synsem local category subj rest> = end
<x0 synsem local content restr rest first> = <x1 synsem local content>.

%%   the ice covered road
%%       ===========
adj -> n v:
phon2
context2
qip2
<x0 dtrs one> = <x1>
<x0 dtrs two> = <x2>
<x0 synsem local category head maj> = <x0 maj>
<x2 synsem local category head form> = ppart
<x0 synsem local category head mod synsem local content> =
    <x2 synsem local category comps first synsem local content>
<x2 synsem local category subj first> = <x1 synsem>
<x0 synsem local category bar> = zero
```

```
<x1 synsem local category bar> = zero
<x0 synsem local content index> =
    <x0 synsem local category head mod synsem local content index>
<x0 synsem local content restr first> =
    <x0 synsem local category head mod synsem local content restr first>
<x0 synsem local content restr rest first> = <x2 synsem local content>
<x0 synsem local content restr rest rest> =
    <x0 synsem local category head mod synsem local content restr rest>.


%% Measure and degree phrases are analyzed as NPs used as adverbs,
%% which then get used as specifiers, of v, adj or adv.

adv -> n:
<x0 phon> = <x1 phon>
<x0 dtrs unarydtr> = <x1>
<x0 synsem local category head maj> = <x0 maj>
<x0 synsem local category head mod> = nil
<x0 synsem local category head spec synsem local category head maj> = v
<x0 synsem local category head spec synsem local category comps> = end
<x0 synsem local category comps> = end
<x1 synsem local category comps> = end
<x0 synsem local category bar> = zero
<x1 synsem local category bar> = two
<x1 synsem local category marking> = unmarked
<x0 synsem local content> = <x1 synsem local content>
<x0 synsem local context> = <x1 synsem local context>
<x0 qstore> = <x1 qstore>.


%% Possessives
det -> n:
<x0 phon> = <x1 phon>
<x0 dtrs unarydtr> = <x1>
<x0 synsem local category head maj> = <x0 maj>
<x1 synsem local category head case> = gen
<x0 synsem local category head spec synsem local category head maj> = n
<x1 synsem local category comps> = end
<x1 synsem local category bar> = two
<x0 synsem local content> = <x1 synsem local content>
<x0 synsem local context> = <x1 synsem local context>
<x0 qstore first det> = definite
<x0 qstore first restind index> =
    <x0 synsem local category head spec synsem local content index>
```

```
<x0 qstore first restind restr first relation> = poss
<x0 qstore first restind restr first possessor> =
    <x0 synsem local content index>
<x0 qstore first restind restr first possessed> =
    <x0 synsem local category head spec synsem local content index>
<x0 qstore first restind restr rest> =
    <x0 synsem local category head spec synsem local content restr>
<x0 qstore rest> = end.
```

A.4 Lexical Entries

```
%% Have to make lex_rule dynamic so new words can be learned.
#copy
:- dynamic lex_rule/1.
#end
```

A.4.1 Definitions that Cut across Syntactic Categories

```
define word:
<phon list rest> = <phon eol>
<synsem local category head maj> = <maj>.

define function:
word
<synsem local category bar> = nil
<synsem local conj> = nil.

define subst:
word
<synsem local category bar> = zero
<synsem local conj> = nil.

define transitive-n:
<synsem local category comps first synsem local category head maj> = n
<synsem local category comps first synsem local category head form> = norm
<synsem local category comps first synsem local category head case> = acc
<synsem local category comps first synsem local category comps> = end
<synsem local category comps first synsem local category bar> = two
<synsem local category comps rest> = end.

define transitive-v-inf:
```

```
<synsem local category comps first synsem local category head maj> = v
<synsem local category comps first synsem local category head form> =
    infinitive
<synsem local category comps first synsem local category comps> = end
<synsem local category comps first synsem local category subj first
    synsem local category head maj> = n
<synsem local category comps first synsem local category subj rest> = end
<synsem local category comps first synsem local category bar> = two
<synsem local category comps rest> = end.

define transitive-s:
<synsem local category comps first synsem local category head maj> = v
<synsem local category comps first synsem local category comps> = end
<synsem local category comps first synsem local category subj> = end
<synsem local category comps rest> = end.

define transitive-adj:
<synsem local category comps first synsem local category head maj> = adj
<synsem local category comps first synsem local category comps> = end
<synsem local category comps first synsem local category bar> = two
<synsem local category comps rest> = end.
```

A.4.2 Adjectives

```
define adjective:
subst
<synsem local category head maj> = adj
<synsem local category head mod synsem local category head maj> = n
<synsem local content index> =
    <synsem local category head mod synsem local content index>
<synsem local content restr first> =
    <synsem local category head mod synsem local content restr first>
<synsem local content restr rest rest> =
    <synsem local category head mod synsem local content restr rest>
<synsem local context list> = <synsem local context eol>
<qstore list> = <qstore eol>.

define modifyingadj:
adjective
<synsem local category subj> = end.

define predicativeadj:
adjective
```

```
<synsem local category subj first synsem> =
    <synsem local category head mod synsem>.


Word here:
predicativeadj
<phon list first> = here
<synsem local category comps> = end
<synsem local content restr rest first arg> =
    <synsem local category head mod synsem local content>
<synsem local content restr rest first relation> = here.

Word old:
adjective
<phon list first> = old
<synsem local category comps> = end
<synsem local content restr rest first arg> =
    <synsem local category head mod synsem local content>
<synsem local content restr rest first relation> = old.

Word righthand:
modifyingadj
<phon list first> = righthand
<synsem local category comps> = end
<synsem local content restr rest first relation> = righthand
<synsem local content restr rest first arg> =
    <synsem local category head mod synsem local content>.

%% Adjectives that take complements

Word afraid:
predicativeadj
transitive-s
<phon list first> = afraid
<synsem local category comps first synsem local category head form> = finite
<synsem local content restr rest first relation> = afraid
<synsem local content restr rest first fearer> =
    <synsem local category head mod synsem local content>
<synsem local content restr rest first feared> =
    <synsem local category comps first synsem local content>.

Word unable:
predicativeadj
transitive-v-inf
```

```
<phon list first> = unable
<synsem local category head mod synsem local content> =
    <synsem local category comps first synsem local category subj first
    synsem local content>
<synsem local content restr rest first relation> = unable
<synsem local content restr rest first unabler> =
    <synsem local category head mod synsem local content>
<synsem local content restr rest first unablee> =
    <synsem local category comps first synsem local content>.
```

A.4.3 Adjectival Prepositions

```
define adjectivaltransitive-n:
predicativeadj
transitive-n
<synsem local content restr rest first arg> =
    <synsem local category head mod synsem local content>
<synsem local content restr rest first comparator> =
    <synsem local category comps first synsem local content>.

Word across:
adjectivaltransitive-n
<phon list first> = across
<synsem local content restr rest first relation> = across.

Word on:
adjectivaltransitive-n
<phon list first> = on
<synsem local content restr rest first relation> = loc.

Word on:
adjectivaltransitive-n
<phon list first> = on
<synsem local content restr rest first relation> = path_change.

Word to:
adjectivaltransitive-n
<phon list first> = to
<synsem local content restr rest first relation> = dest.

Word to:
adjectivaltransitive-n
<phon list first> = to
```

```
<synsem local content restr rest first relation> = dir.

Word toward:
adjectivaltransitive-n
<phon list first> = toward
<synsem local content restr rest first relation> = dest.

Word up to:
adjectivaltransitive-n
<phon list first> = upto
<synsem local content restr rest first relation> = dest.

%% (Subject) Relative pronouns are treated as adjectives that
%% subcategorize for a verb phrase

define relpro:
subst
<synsem local category head maj> = adj
<synsem local category head mod synsem local category head maj> = n
<synsem local category head mod synsem local content> =
    <synsem local category comps first synsem local category subj first
    synsem local content>
<synsem local category comps first synsem local category head maj> = v
<synsem local category comps first synsem local category head form> = finite
<synsem local category comps first synsem local category subj first
    synsem local category head maj> = n
<synsem local category comps first synsem local category subj first
    synsem local category head case> = nom
<synsem local category comps first synsem local category subj first
    synsem local category head form> = norm
<synsem local category comps first synsem local category subj rest> = end
<synsem local category comps first synsem local category comps> = end
<synsem local category comps rest> = end
<synsem local content index> = <synsem local category head mod
    synsem local content index>
<synsem local content restr first> = <synsem local category head mod
    synsem local content restr first>
<synsem local content restr rest first> = <synsem local category comps first
    synsem local content>
<synsem local content restr rest rest> = <synsem local category head mod
    synsem local content restr rest>
<qstore list> = <qstore eol>.

Word who:
```

```
relpro
<phon list first> = which.

Word which:
relpro
<phon list first> = which.

Word that:
relpro
<phon list first> = that.
```

A.4.4 Adverbials

A.4.4.1 Complement Adverbs

```
%% Intransitive complement adverbs

define adverb-comp:
subst
<synsem local category head maj> = adv
<synsem local category head mod> = nil
<synsem local category head spec> = nil
<synsem local category comps> = end
<synsem local context list> = <synsem local context eol>
<qstore list> = <qstore eol>.

Word straight:
adverb-comp
<phon list first> = straight
<synsem local content relation> = straight.

Word south:
adverb-comp
<phon list first> = south
<synsem local content relation> = south.

%% A particle is an intransitive adverb complement.

Word on:
adverb-comp
<phon list first> = on
<synsem local content relation> = on.
```

```
Word up:
adverb-comp
<phon list first> = up
<synsem local content relation> = up.

%% Transitive complement adverbs

define adverbial-transitive-ncomp:
subst
transitive-n
<synsem local category head maj> = adv
<synsem local category head mod> = nil
<synsem local category head spec> = nil
<synsem local content comparator> =
    <synsem local category comps first synsem local content>
<synsem local context list> = <synsem local context eol>
<qstore list> = <qstore eol>.

Word across:
adverbial-transitive-ncomp
<phon list first> = across
<synsem local content relation> = across.

Word on:
adverbial-transitive-ncomp
<phon list first> = on
<synsem local content relation> = loc.

Word on:
adverbial-transitive-ncomp
<phon list first> = on
<synsem local content relation> = path_change.

Word to:
adverbial-transitive-ncomp
<phon list first> = to
<synsem local content relation> = dest.

Word to:
adverbial-transitive-ncomp
<phon list first> = to
<synsem local content relation> = dir.
```

A.4.4.2 Modifying Adverbs

```
%% Intransitive modifying adverbs

define adverb-mod-vp:
subst
<synsem local category head maj> = adv
<synsem local category head mod synsem local category head maj> = v
<synsem local category head mod synsem local category head form> =
    {base finite ppart prog}
<synsem local category head mod synsem local category subj first
    synsem local category head maj> = n
<synsem local category head mod synsem local category subj rest> = end
<synsem local content psoa> =
    <synsem local category head mod synsem local content psoa>
<synsem local content restr first soaarg> =
    <synsem local category head mod synsem local content psoa>
<synsem local content restr rest> =
    <synsem local category head mod synsem local content restr>
<synsem local context list> = <synsem local context eol>
<qstore list> = <qstore eol>.

define intrans-adverb-mod-vp:
adverb-mod-vp
<synsem local category comps> = end.

Word slightly:
intrans-adverb-mod-vp
<phon list first> = slightly
<synsem local content restr first relation> = slightly.

Word straight:
intrans-adverb-mod-vp
<phon list first> = straight
<synsem local content restr first relation> = straight.

Word here:
intrans-adverb-mod-vp
<phon list first> = here
<synsem local content restr first relation> = here.

%% Sentence-modifying adverbs
```

```
define adverb-mod-s:
subst
<synsem local category head maj> = adv
<synsem local category head mod synsem local category head maj> = v
<synsem local category head mod synsem local category subj> = end
<synsem local category head mod synsem local category comps> = end
<synsem local category head mod synsem local category bar> = two
<synsem local content psoa> =
    <synsem local category head mod synsem local content psoa>
<synsem local content restr first soaarg> =
    <synsem local category head mod synsem local content psoa>
<synsem local content restr rest> =
    <synsem local category head mod synsem local content restr>
<synsem local context list> = <synsem local context eol>
<qstore list> = <qstore eol>.

define intrans-adverb-mod-s:
adverb-mod-s
<synsem local category comps> = end.

Word here:
intrans-adverb-mod-s
<phon list first> = here
<synsem local content restr first relation> = here.

Word then:
intrans-adverb-mod-s
<phon list first> = then
<synsem local content restr first relation> = then.

Word supposedly:
intrans-adverb-mod-s
<phon list first> = supposedly
<synsem local content restr first relation> = supposedly.


%% Transitive modifying adverbs

define adverbial-transitive-n-mod-vp:
adverb-mod-vp
transitive-n
<synsem local content restr first comparator> =
    <synsem local category comps first synsem local content>.
```

BEST COPY AVAILABLE

```
Word across:
adverbial-transitive-n-mod-vp
<phon list first> = across
<synsem local content restr first relation> = across.

Word by:
adverbial-transitive-n-mod-vp
<phon list first> = by
<synsem local content restr first relation> = means.

Word on:
adverbial-transitive-n-mod-vp
<phon list first> = on
<synsem local content restr first relation> = loc.

Word on:
adverbial-transitive-n-mod-vp
<phon list first> = on
<synsem local content restr first relation> = path_change.

Word to:
adverbial-transitive-n-mod-vp
<phon list first> = to
<synsem local content restr first relation> = dest.

Word to:
adverbial-transitive-n-mod-vp
<phon list first> = to
<synsem local content restr first relation> = dir.

Word toward:
adverbial-transitive-n-mod-vp
<phon list first> = toward
<synsem local content restr first relation> = toward.


define adverbial-transitive-n-mod-s:
adverb-mod-s
transitive-n
<synsem local content restr first restr> = end.

Word according to:
adverbial-transitive-n-mod-s
```

```
<phon list first> = accordingto
<synsem local content restr first relation> = accordingto
<synsem local content restr first soaarg> =
    <synsem local category head mod synsem local content psoa>
<synsem local content restr first comparator> =
    <synsem local category comps first synsem local content>.

Word after:
adverbial-transitive-n-mod-s
<phon list first> = after
<synsem local content restr first relation> = after
<synsem local content restr first soaarg> =
    <synsem local category head mod synsem local content psoa>
<synsem local content restr first comparator> =
    <synsem local category comps first synsem local content>.

Word simultaneous with:
adverbial-transitive-n-mod-s
<phon list first> = simultaneouswith
<synsem local content restr first relation> = simultaneouswith
<synsem local content restr first soaarg> =
    <synsem local category head mod synsem local content psoa>
<synsem local content restr first comparator> =
    <synsem local category comps first synsem local content>.

%% Treating this as a ditransitive vp-modifying adverb
%%     Stopped within a foot of a utility pole.

Word within:
adverb-mod-vp
<phon list first> = within
<synsem local category comps first synsem local category head maj> = n
<synsem local category comps first synsem local category head form> = norm
<synsem local category comps first synsem local category head case> = acc
<synsem local category comps first synsem local category comps> = end
<synsem local category comps first synsem local category bar> = two
<synsem local category comps rest first synsem local category head maj> = adv
<synsem local category comps rest first synsem local category comps> = end
<synsem local category comps rest first synsem local category bar> = two
<synsem local category comps rest first synsem local content relation> = of
<synsem local category comps rest rest> = end
<synsem local content restr first relation> = within
<synsem local content restr first soaarg> =
    <synsem local category head mod synsem local content psoa>
```

```
<synsem local content restr first comparator> =
    <synsem local category comps first synsem local content>.
```

A.4.4.3 Adverbial Subordinating Conjunctions

```
define subconj:
transitive-s
<synsem local category comps first synsem local category head form> = finite
<synsem local content psoa> =
    <synsem local category comps first synsem local content psoa>
<synsem local content restr first soaarg> =
    <synsem local category head mod synsem local content psoa>
<synsem local content restr rest> =
    <synsem local category head mod synsem local content restr>
<synsem local context list> = <synsem local context eol>
<qstore list> = <qstore eol>.

%% Subconj introducing VP-modifying clause

Word because:
adverb-mod-vp
subconj
<phon list first> = because
<synsem local content restr first relation> = because.

Word until:
adverb-mod-vp
subconj
<phon list first> = until
<synsem local content restr first relation> = until.

Word while:
adverb-mod-vp
subconj
<phon list first> = while
<synsem local content restr first relation> = while.

%% Subconj introducing S-modifying clause

Word until:
adverb-mod-s
subconj
<phon list first> = until
```

```
<synsem local content restr first relation> = until.

Word when:
adverb-mod-s
subconj
<phon list first> = when
<synsem local content restr first relation> = when.
```

A.4.4.4 Specifying Adverbs

```
define adverb-spec-v:
subst
<synsem local category head maj> = adv
<synsem local category head mod> = nil
<synsem local category head spec synsem local category head maj> = v
<synsem local category comps> = end
<synsem local context list> = <synsem local context eol>
<qstore first> = <synsem local content>
<qstore rest> = end.

Word some:
adverb-spec-v
<phon list first> = some
<synsem local content degree> = some
<synsem local content restind> =
    <synsem local category head spec synsem local content>.

define adverb-spec-adj:
subst
<synsem local category head maj> = adv
<synsem local category head mod> = nil
<synsem local category head spec synsem local category head maj> = adj
<synsem local category comps> = end
<synsem local context list> = <synsem local context eol>
<qstore first> = <synsem local content>
<qstore rest> = end.

Word very:
adverb-spec-adj
<phon list first> = very
<synsem local content degree> = very
<synsem local content restind> =
    <synsem local category head spec synsem local content>.
```

```
%% He was not happy.
Word not:
adverb-spec-adj
<phon list first> = not
<synsem local content polarity> = not
<synsem local content restind> =
    <synsem local category head spec synsem local content>.

define adverb-spec-adv:
subst
<synsem local category head maj> = adv
<synsem local category head mod> = nil
<synsem local category head spec synsem local category head maj> = adv
<synsem local category head spec synsem local category comps> = end
<synsem local category comps> = end
<synsem local context list> = <synsem local context eol>
<qstore first> = <synsem local content>
<qstore rest> = end.

Word all the way:
adverb-spec-adv
<phon list first> = alltheway
<synsem local content degree> = alltheway
<synsem local content restind> =
    <synsem local category head spec synsem local content>.

Word just:
adverb-spec-adv
<phon list first> = just
<synsem local content degree> = just
<synsem local content restind> =
    <synsem local category head spec synsem local content>.
```

A.4.5 Conjunctions

```
Word both:
word
<phon list first> = both
<synsem local category head maj> = conj
<synsem local conj> = both
<synsem local content> = both
<synsem local context list> = <synsem local context eol>
```

```
<qstore list> = <qstore eol>.

Word and:
word
<phon list first> = and
<synsem local category head maj> = conj
<synsem local conj> = and
<synsem local content> = and
<synsem local context list> = <synsem local context eol>
<qstore list> = <qstore eol>.

Word but:
word
<phon list first> = but
<synsem local category head maj> = conj
<synsem local content> = but
<synsem local context list> = <synsem local context eol>
<qstore list> = <qstore eol>.
```

A.4.6 Determiners

```
define determiner:
function
<synsem local category head maj> = det
<synsem local category head spec synsem local category head maj> = n.

Word a:
determiner
<phon list first> = a
<synsem local category head spec synsem local content index num> = singular
<synsem local content det> = indefinite
<synsem local content restind> =
    <synsem local category head spec synsem local content>
<synsem local context list> = <synsem local context eol>
<qstore list first> = <synsem local content>
<qstore list rest> = <qstore eol>.

Word the:
determiner
<phon list first> = the
<synsem local content det> = definite
<synsem local content restind> =
    <synsem local category head spec synsem local content>
```

```
<synsem local context list> = <synsem local context eol>
<qstore list first> = <synsem local content>
<qstore list rest> = <qstore eol>.

Word no:
determiner
<phon list first> = no
<synsem local content det> = no
<synsem local content restind> =
    <synsem local category head spec synsem local content>
<synsem local context list> = <synsem local context eol>
<qstore list first> = <synsem local content>
<qstore list rest> = <qstore eol>.

Word another:
determiner
<phon list first> = another
<synsem local content det> = indefinite
<synsem local content restind> =
    <synsem local category head spec synsem local content>
<synsem local context list> = <synsem local context eol>
<qstore list first> = <synsem local content>
<qstore list rest> = <qstore eol>.

Word my:
determiner
<phon list first> = my
<synsem local content index per> = first
<synsem local content index num> = singular
<synsem local context list first cindices speaker> =
    <synsem local content index>
<synsem local context list rest> = <synsem local context eol>
<qstore list first det> = definite
<qstore list first restind index> =
    <synsem local category head spec synsem local content index>
<qstore list first restind restr first relation> = poss
<qstore list first restind restr first possessor> =
    <synsem local content index>
<qstore list first restind restr first possessed> =
    <synsem local category head spec synsem local content index>
<qstore list first restind restr rest> =
    <synsem local category head spec synsem local content restr>
<qstore list rest> = <qstore eol>.
```

BEST COPY AVAILABLE

```
Word her:
determiner
<phon list first> = her
<synsem local content index per> = third
<synsem local content index num> = singular
<synsem local content index gend> = fem
<synsem local context list> = <synsem local context eol>
<qstore list first det> = definite
<qstore list first restind index> =
    <synsem local category head spec synsem local content index>
<qstore list first restind restr first relation> = poss
<qstore list first restind restr first possessor> =
    <synsem local content index>
<qstore list first restind restr first possessed> =
    <synsem local category head spec synsem local content index>
<qstore list first restind restr rest> =
    <synsem local category head spec synsem local content restr>
<qstore list rest> = <qstore eol>.

Word numis:
determiner
<phon list first> = numis
<synsem local content index> = ref
<synsem local context list first backgr first relation> = naming
<synsem local context list first backgr first bearer> =
    <synsem local content index>
<synsem local context list first backgr first name> = numi
<synsem local context list rest> = <synsem local context eol>
<qstore list first det> = definite
<qstore list first restind index> =
    <synsem local category head spec synsem local content index>
<qstore list first restind restr first relation> = poss
<qstore list first restind restr first possessor> =
    <synsem local content index>
<qstore list first restind restr first possessed> =
    <synsem local category head spec synsem local content index>
<qstore list first restind restr rest> =
    <synsem local category head spec synsem local content restr>
<qstore list rest> = <qstore eol>.
```

A.4.7 Markers

```
Word that:
word
<phon list first> = that
<synsem local category head maj> = comp
<synsem local category head spec synsem local category head maj> = v
<synsem local category head spec synsem local category head form> =
    {base finite}
<synsem local category head spec synsem local category subj> = end
<synsem local category head spec synsem local category comps> = end
<synsem local category head spec synsem local category marking> = unmarked
<synsem local category comps> = end
<synsem local category marking> = that
<synsem local context list> = <synsem local context eol>
<qstore list> = <qstore eol>.

Word by:
word
<phon list first> = by
<synsem local category head maj> = comp
<synsem local category head spec synsem local category head maj> = n
<synsem local category head spec synsem local category head case> = acc
<synsem local category head spec synsem local category subj> = end
<synsem local category head spec synsem local category comps> = end
<synsem local category head spec synsem local category bar> = two
<synsem local category head spec synsem local category marking> = unmarked
<synsem local category head case> = acc
<synsem local category comps> = end
<synsem local category marking> = by
<synsem local context list> = <synsem local context eol>
<qstore list> = <qstore eol>.

Word to:
word
<phon list first> = to
<synsem local category head maj> = comp
<synsem local category head spec synsem local category head maj> = n
<synsem local category head spec synsem local category head case> = acc
<synsem local category head spec synsem local category subj> = end
<synsem local category head spec synsem local category comps> = end
<synsem local category head spec synsem local category bar> = two
<synsem local category head spec synsem local category marking> = unmarked
```

```
<synsem local category head case> = acc
<synsem local category comps> = end
<synsem local category marking> = to
<synsem local context list> = <synsem local context eol>
<qstore list> = <qstore eol>.
```

A.4.8 Nouns

```
define propernoun:
subst
<synsem local category head maj> = n
<synsem local category head form> = norm
<synsem local category head case> = {acc nom}
<synsem local category comps> = end
<synsem local category spr> = end
<synsem local category marking> = unmarked
<synsem local context list first backgr first relation> = naming
<synsem local context list first backgr first bearer> =
    <synsem local content index>
<synsem local context list rest> = <synsem local context eol>
<qstore list> = <qstore eol>.

define propernoun3sg:
propernoun
<synsem local content index gend> = neuter
<synsem local content index per> = third
<synsem local content index num> = singular.

%% Vehicles

Word numi:
propernoun3sg
<phon list first> = numi
<synsem local context backgr first name> = numi.

%% People

Word edward ladewick:
propernoun3sg
<phon list first> = 'Edward Ladewick'
<synsem local context backgr first name> = edwardladewick.

Word meuser:
```

```
propernoun3sg
<phon list first> = 'meuser'
<synsem local context backgr first name> = meuser.

%% Streets

Word bark wood rd:
propernoun3sg
<phon list first> = 'Bark Wood Rd'
<synsem local context backgr first name> = barkwoodrd.

Word university avenue:
propernoun3sg
<phon list first> = 'University Avenue'
<synsem local context backgr first name> = universityavenue.

Word elm:
propernoun3sg
<phon list first> = 'Elm St'
<synsem local context backgr first name> = elm.

Word kirby:
propernoun3sg
<phon list first> = kirby
<synsem local context backgr first name> = kirby.

%% Interstates
Word i74:
propernoun3sg
<phon list first> = i74
<synsem local context backgr first name> = i74.

Word interstate57:
propernoun3sg
<phon list first> = interstate57
<synsem local context backgr first name> = interstate57.

%% Buildings

Word beckman:
propernoun3sg
<phon list first> = beckman
<synsem local context backgr first name> = beckman.
```

```
Word saint anthony hospital:
propernoun3sg
<phon list first> = 'Saint Anthony Hospital'
<synsem local context backgr first name> = saintanthonyhospital.

%% Towns

Word effingham:
propernoun3sg
<phon list first> = effingham
<synsem local context backgr first name> = effingham.

Word urbana:
propernoun3sg
<phon list first> = urbana
<synsem local context backgr first name> = urbana.

%% Companies

Word howell asphalt:
propernoun3sg
<phon list first> = howellasphalt
<synsem local context backgr first name> = howellasphalt.

% common nouns

define commonnoun:
subst
<synsem local category head maj> = n
<synsem local category head form> = norm
<synsem local category head case> = {acc nom}
<synsem local category marking> = unmarked
<synsem local content index gend> = neuter
<synsem local content index per> = third
<synsem local content restr first instance> = <synsem local content index>
<synsem local content restr rest> = end
<synsem local context list> = <synsem local context eol>
<qstore list> = <qstore eol>.

define gencommonnoun:
subst
<synsem local category head maj> = n
<synsem local category head form> = norm
<synsem local category head case> = gen
```

BEST COPY AVAILABLE

```
<synsem local category marking> = unmarked
<synsem local content index gend> = neuter
<synsem local content index per> = third
<synsem local content restr first instance> = <synsem local content index>
<synsem local content restr rest> = end.


Word accident:
commonnoun
<phon list first> = accident
<synsem local category spr first synsem local category head maj> = det
<synsem local category spr rest> = end
<synsem local content index num> = singular
<synsem local content restr first relation> = accident.

Word block:
commonnoun
<phon list first> = block
<synsem local content index num> = singular
<synsem local content restr first relation> = block.

Word blocks:
commonnoun
<phon list first> = blocks
<synsem local content index num> = plural
<synsem local content restr first relation> = block.

Word fender:
commonnoun
<phon list first> = fender
<synsem local category spr first synsem local category head maj> = det
<synsem local category spr rest> = end
<synsem local content index num> = singular
<synsem local content restr first relation> = fender.

Word left:
commonnoun
<phon list first> = left
<synsem local category spr first synsem local category head maj> = det
<synsem local category spr rest> = end
<synsem local content index num> = singular
<synsem local content restr first relation> = left.

Word people:
```

```
commonnoun
<phon list first> = people
<synsem local content index num> = plural
<synsem local content restr first relation> = people.

%% pronouns

Word i:
subst
<phon list first> = i
<synsem local category head maj> = n
<synsem local category head form> = norm
<synsem local category head case> = nom
<synsem local category spr> = end
<synsem local content index per> = first
<synsem local content index num> = singular
<synsem local context list first cindices speaker> =
    <synsem local content index>
<synsem local context list rest> = <synsem local context eol>
<qstore list> = <qstore eol>.

Word me:
subst
<phon list first> = me
<synsem local category head maj> = n
<synsem local category head form> = norm
<synsem local category head case> = acc
<synsem local category spr> = end
<synsem local content index per> = first
<synsem local content index num> = singular
<synsem local context list first cindices speaker> =
    <synsem local content index>
<synsem local context list rest> = <synsem local context eol>
<qstore list> = <qstore eol>.

Word you:
subst
<phon list first> = you
<synsem local category head maj> = n
<synsem local category head form> = norm
<synsem local category head case> = nom
<synsem local category spr> = end
<synsem local content index per> = second
<synsem local context list first cindices addressee> = <synsem local index>
```

```
<synsem local context list rest> = <synsem local context eol>
<qstore list> = <qstore eol>.

Word she:
subst
<phon list first> = she
<synsem local category head maj> = n
<synsem local category head form> = norm
<synsem local category head case> = nom
<synsem local category spr> = end
<synsem local content index per> = third
<synsem local content index num> = singular
<synsem local content index gend> = fem
<synsem local context list first backgr first relation> = female
<synsem local context list first backgr first instance> =
    <synsem local content index>
<synsem local context list rest> = <synsem local context eol>
<qstore list> = <qstore eol>.

%% Expletives

Word there:
subst
<phon list first> = there
<synsem local category head maj> = n
<synsem local category head form> = there
<synsem local category spr> = end
<synsem local content index per> = third
<synsem local context list> = <synsem local context eol>
<qstore list> = <qstore eol>.

Word it:
subst
<phon list first> = it
<synsem local category head maj> = n
<synsem local category head form> = it
<synsem local category spr> = end
<synsem local content index per> = third
<synsem local context list> = <synsem local context eol>
<qstore list> = <qstore eol>.
```

A.4.9 Verbs

```
define verb:
subst
<synsem local category head maj> = v
<synsem local category subj first synsem local category head maj> = n
<synsem local category subj first synsem local category head case> = nom
<synsem local category subj first synsem local category head form> = norm
<synsem local category subj first synsem local category comps> = end
<synsem local category subj first synsem local category subj> = end
<synsem local category subj first synsem local category spr> = end
<synsem local category subj first synsem local category bar> = two
<synsem local category subj first synsem local conj> = nil
<synsem local category subj rest> = end
<synsem local content restr> = end
<synsem local context list> = <synsem local context eol>
<qstore list> = <qstore eol>.

define auxverb:
verb
<synsem local category comps first synsem local category head maj> = v
<synsem local category comps first synsem local category subj first
    synsem local category head maj> = n
<synsem local category comps first synsem local category subj rest> = end
<synsem local category comps first synsem local category bar> = two
<synsem local category comps rest> = end
<synsem local category subj> =
    <synsem local category comps first synsem local category subj>
<synsem local content> =
    <synsem local category comps first synsem local content>.

define modalauxverb:
auxverb
<synsem local category head form> = finite
<synsem local category comps first synsem local category head form> = base.

Word to:
auxverb
<phon list first> = to
<synsem local category head form> = infinitive
<synsem local category comps first synsem local category head form> = base.

Word does:
```

```
modalauxverb
<phon list first> = did
<synsem local category subj first synsem local content index per> = third
<synsem local category subj first synsem local content index num> = singular
<synsem local content tense> = present.

Word did:
modalauxverb
<phon list first> = did
<synsem local content tense> = past.

Word d:
modalauxverb
<phon list first> = d
<synsem local content modal> = would.

Word could:
modalauxverb
<phon list first> = could
<synsem local content modal> = could.

Word ll:
modalauxverb
<phon list first> = ll
<synsem local content tense> = future.

define verb-compvinf:
verb
transitive-v-inf
<synsem local category subj> =
    <synsem local category comps first synsem local category subj>.

Word re going:
verb-compvinf
<phon list first> = regoing
<synsem local category head form> = finite
<synsem local content> =
    <synsem local category comps first synsem local content>
<synsem local content time> = near_future.

define verb-compvppart:
verb
<synsem local category comps first synsem local category head maj> = v
<synsem local category comps first synsem local category head form> = ppart
```

```
<synsem local category comps first synsem local category subj first
    synsem local category head maj> = n
<synsem local category comps first synsem local category subj rest> = end
<synsem local category comps first synsem local category bar> = two
<synsem local category comps rest> = end
<synsem local category subj> =
    <synsem local category comps first synsem local category subj>.

define verb-compadv:
verb
<synsem local category comps first synsem local category head maj> = adv
<synsem local category comps first synsem local category head mod> = nil
<synsem local category comps first synsem local category head spec> = nil
<synsem local category comps first synsem local category comps> = end
<synsem local category comps first synsem local category bar> = two
<synsem local category comps rest> = end.

define verb-compn:
verb
transitive-n
<synsem local content restr> = end.

define verb-comps:
verb
transitive-s
<synsem local content restr> = end.

define verb-compadj:
verb
transitive-adj
<synsem local category subj first synsem> = <synsem local category
comps first synsem local category head mod synsem>
<synsem local category comps first synsem local category subj first synsem> =
    <synsem local category comps first synsem local category head mod synsem>.

define verbintrans:
verb
<synsem local category comps> = end.


Word approaching:
verbintrans
<phon list first> = approaching
<synsem local category head form> = prog
```

```
<synsem local content psoa relation> = approach
<synsem local content psoa approacher> =
    <synsem local category subj first synsem local content>.

Word approaching:
verb-compn
<phon list first> = approaching
<synsem local category head form> = prog
<synsem local content psoa relation> = approach
<synsem local content psoa approached> =
    <synsem local category comps first synsem local content>
<synsem local content psoa approacher> =
    <synsem local category subj first synsem local content>.

Word beat:
verb-compn
<phon list first> = beat
<synsem local category head form> = base
<synsem local content psoa relation> = beat
<synsem local content psoa beatee> =
    <synsem local category comps first synsem local content>
<synsem local content psoa beater> =
    <synsem local category subj first synsem local content>.

Word beat:
verb-compn
<phon list first> = beat
<synsem local category head form> = finite
<synsem local category subj first synsem local content index per> =
    {first second}
<synsem local category subj first synsem local content index num> = singular
<synsem local content tense> = present
<synsem local content psoa relation> = beat
<synsem local content psoa beatee> =
    <synsem local category comps first synsem local content>
<synsem local content psoa beater> =
    <synsem local category subj first synsem local content>.

Word beat:
verb-compn
<phon list first> = beat
<synsem local category head form> = finite
<synsem local category subj first synsem local content index num> = plural
<synsem local content tense> = present
```

```
<synsem local content psoa relation> = beat
<synsem local content psoa beatee> =
    <synsem local category comps first synsem local content>
<synsem local content psoa beater> =
    <synsem local category subj first synsem local content>.

Word beat:
verb-compn
<phon list first> = beat
<synsem local category head form> = finite
<synsem local content tense> = past
<synsem local content psoa relation> = beat
<synsem local content psoa beatee> =
    <synsem local category comps first synsem local content>
<synsem local content psoa beater> =
    <synsem local category subj first synsem local content>.

Word bent:
verb-compn
<phon list first> = bent
<synsem local category head form> = ppart
<synsem local content psoa relation> = bend
<synsem local content psoa bendee> =
    <synsem local category comps first synsem local content>
<synsem local content psoa bender> =
    <synsem local category subj first synsem local content>.

Word carrying:
verb-compn
<phon list first> = carrying
<synsem local category head form> = prog
<synsem local content psoa relation> = carry
<synsem local content psoa carried> =
    <synsem local category comps first synsem local content>
<synsem local content psoa carrier> =
    <synsem local category subj first synsem local content>.

Word dodge:
verbintrans
<phon list first> = dodge
<synsem local category head form> = base
<synsem local content psoa relation> = dodge
<synsem local content psoa dodger> =
    <synsem local category subj first synsem local content>.
```

BEST COPY AVAILABLE

```
Word dodge:
verbintrans
<phon list first> = dodge
<synsem local category head form> = finite
<synsem local category subj first synsem local content index per> =
    {first second}
<synsem local category subj first synsem local content index num> = singular
<synsem local content tense> = present
<synsem local content psoa relation> = dodge
<synsem local content psoa dodger> =
    <synsem local category subj first synsem local content>.

Word dodge:
verbintrans
<phon list first> = dodge
<synsem local category head form> = finite
<synsem local category subj first synsem local content index num> = plural
<synsem local content tense> = present
<synsem local content psoa relation> = dodge
<synsem local content psoa dodger> =
    <synsem local category subj first synsem local content>.

Word flipped:
verb-compadv
<phon list first> = flipped
<synsem local category head form> = finite
<synsem local content tense> = past
<synsem local content psoa relation> = flip
<synsem local content psoa loc> =
    <synsem local category comps first synsem local content>
<synsem local content psoa flipper> =
    <synsem local category subj first synsem local content>.

Word gave:
verb
<phon list first> = gave
<synsem local category head form> = finite
<synsem local category comps first synsem local category head maj> = n
<synsem local category comps first synsem local category head case> = acc
<synsem local category comps first synsem local category head form> = norm
<synsem local category comps first synsem local category bar> = two
<synsem local category comps first synsem local category comps> = end
<synsem local category comps rest first synsem local category head maj> = n
```

```
<synsem local category comps rest first synsem local category head case> = acc
<synsem local category comps rest first synsem local category head form> = norm
<synsem local category comps rest first synsem local category comps> = end
<synsem local category comps rest rest> = end
<synsem local content tense> = past
<synsem local content psoa relation> = give
<synsem local content psoa object> =
    <synsem local category comps rest first synsem local content>
<synsem local content psoa recipient> =
    <synsem local category comps first synsem local content>
<synsem local content psoa giver> =
    <synsem local category subj first synsem local content>.

Word get:
verb-compadj
<phon list first> = get
<synsem local category head form> = base
<synsem local content psoa relation> = become
<synsem local content psoa quality> =
    <synsem local category comps first synsem local content restr rest first>
<synsem local content psoa becomer> =
    <synsem local category subj first synsem local content>.

Word get:
verb-compadj
<phon list first> = get
<synsem local category head form> = finite
<synsem local category subj first synsem local content index per> =
    {first second}
<synsem local category subj first synsem local content index num> = singular
<synsem local content tense> = present
<synsem local content psoa relation> = become
<synsem local content psoa quality> =
    <synsem local category comps first synsem local content restr rest first>
<synsem local content psoa becomer> =
    <synsem local category subj first synsem local content>.

Word get:
verb-compadj
<phon list first> = get
<synsem local category head form> = finite
<synsem local category subj first synsem local content index num> = plural
<synsem local content tense> = present
<synsem local content psoa relation> = become
```

```
<synsem local content psoa quality> =
    <synsem local category comps first synsem local content restr rest first>
<synsem local content psoa becomer> =
    <synsem local category subj first synsem local content>.

Word get:
verb-compadv
<phon list first> = get
<synsem local category head form> = base
<synsem local content psoa relation> = arrive
<synsem local content psoa arriver> =
    <synsem local category subj first synsem local content>
<synsem local content psoa loc> =
    <synsem local category comps first synsem local content>.

Word hit:
verb-compn
<phon list first> = hit
<synsem local category head form> = base
<synsem local content psoa relation> = hit
<synsem local content psoa hittee> =
    <synsem local category comps first synsem local content>
<synsem local content psoa hitter> =
    <synsem local category subj first synsem local content>.

Word hit:
verb-compn
<phon list first> = hit
<synsem local category head form> = finite
<synsem local category subj first synsem local content index per> =
    {first second}
<synsem local category subj first synsem local content index num> = singular
<synsem local content tense> = present
<synsem local content psoa relation> = hit
<synsem local content psoa hittee> =
    <synsem local category comps first synsem local content>
<synsem local content psoa hitter> =
    <synsem local category subj first synsem local content>.

Word hit:
verb-compn
<phon list first> = hit
<synsem local category head form> = finite
<synsem local category subj first synsem local content index num> = plural
```

```
<synsem local content tense> = present
<synsem local content psoa relation> = hit
<synsem local content psoa hittee> =
    <synsem local category comps first synsem local content>
<synsem local content psoa hitter> =
    <synsem local category subj first synsem local content>.

Word hit:
verb-compn
<phon list first> = hit
<synsem local category head form> = finite
<synsem local content tense> = past
<synsem local content psoa relation> = hit
<synsem local content psoa hittee> =
    <synsem local category comps first synsem local content>
<synsem local content psoa hitter> =
    <synsem local category subj first synsem local content>.

Word hit:
verb-compn
<phon list first> = hit
<synsem local category head form> = ppart
<synsem local content psoa relation> = hit
<synsem local content psoa hittee> =
    <synsem local category comps first synsem local content>
<synsem local content psoa hitter> =
    <synsem local category subj first synsem local content>.

Word hitting:
verb-compn
<phon list first> = hitting
<synsem local category head form> = prog
<synsem local content psoa relation> = hit
<synsem local content psoa hittee> =
    <synsem local category comps first synsem local content>
<synsem local content psoa hitter> =
    <synsem local category subj first synsem local content>.

Word occurred:
verbintrans
<phon list first> = occurred
<synsem local category head form> = finite
<synsem local content tense> = past
<synsem local content psoa relation> = occur
```

```
<synsem local content psoa event> =
    <synsem local category subj first synsem local content>.

Word posted:
verb-compn
<phon list first> = posted
<synsem local category head form> = ppart
<synsem local content psoa relation> = post
<synsem local content psoa postee> =
    <synsem local category comps first synsem local content>
<synsem local content psoa poster> =
    <synsem local category subj first synsem local content>.

Word prepare:
verb-compvinf
<phon list first> = prepare
<synsem local category head form> = base
<synsem local content psoa relation> = prepare
<synsem local content psoa preparer> =
    <synsem local category subj first synsem local content>
<synsem local content psoa prepared> =
    <synsem local category comps first synsem local content>.

Word prepare:
verb-compvinf
<phon list first> = prepare
<synsem local category head form> = finite
<synsem local category subj first synsem local content index per> =
    {first second}
<synsem local category subj first synsem local content index num> = singular
<synsem local content tense> = present
<synsem local content psoa relation> = prepare
<synsem local content psoa preparer> =
    <synsem local category subj first synsem local content>
<synsem local content psoa prepared> =
    <synsem local category comps first synsem local content>.

Word prepare:
verb-compvinf
<phon list first> = prepare
<synsem local category head form> = finite
<synsem local category subj first synsem local content index num> = plural
<synsem local content tense> = present
<synsem local content psoa psoa relation> = prepare
```

```
<synsem local content psoa preparer> =
    <synsem local category subj first synsem local content>
<synsem local content psoa prepared> =
    <synsem local category comps first synsem local content>.

Word ran over:
verb-compn
<phon list first> = ranover
<synsem local category head form> = finite
<synsem local content tense> = past
<synsem local content psoa relation> = runover
<synsem local content psoa runover> =
    <synsem local category comps first synsem local content>
<synsem local content psoa runnerover> =
    <synsem local category subj first synsem local content>.

Word see:
verb-compn
<phon list first> = see
<synsem local category head form> = base
<synsem local content psoa relation> = see
<synsem local content psoa seen> =
    <synsem local category comps first synsem local content>
<synsem local content psoa seer> =
    <synsem local category subj first synsem local content>.

Word see:
verb-compn
<phon list first> = see
<synsem local category head form> = finite
<synsem local category subj first synsem local content index per> =
    {first second}
<synsem local category subj first synsem local content index num> = singular
<synsem local content tense> = present
<synsem local content psoa relation> = see
<synsem local content psoa seen> =
    <synsem local category comps first synsem local content>
<synsem local content psoa seer> =
    <synsem local category subj first synsem local content>.

Word see:
verb-compn
<phon list first> = see
<synsem local category head form> = finite
```

```
<synsem local category subj first synsem local content index num> = plural
<synsem local content tense> = present
<synsem local content psoa relation> = see
<synsem local content psoa seen> =
    <synsem local category comps first synsem local content>
<synsem local content psoa seer> =
    <synsem local category subj first synsem local content>.

Word saw:
verb-compn
<phon list first> = saw
<synsem local category head form> = finite
<synsem local content tense> = past
<synsem local content psoa relation> = see
<synsem local content psoa seen> =
    <synsem local category comps first synsem local content>
<synsem local content psoa seer> =
    <synsem local category subj first synsem local content>.

Word saw:
verb-comps
<phon list first> = saw
<synsem local category head form> = finite
<synsem local category comps first synsem local category head form> = base
<synsem local content tense> = past
<synsem local content psoa relation> = see
<synsem local content psoa seen> =
    <synsem local category comps first synsem local content>
<synsem local content psoa seer> =
    <synsem local category subj first synsem local content>.

%  TAKE1 (as in "take a right")

Word take:
verb-compn
<phon list first> = take
<synsem local category head form> = base
<synsem local content psoa relation> = turn
<synsem local content psoa dir> =
    <synsem local category comps first synsem local content>
<synsem local content psoa turner> =
    <synsem local category subj first synsem local content>.

Word take:
```

```
verb-compn
<phon list first> = take
<synsem local category head form> = finite
<synsem local category subj first synsem local content index per> =
    {first second}
<synsem local category subj first synsem local content index num> = singular
<synsem local content tense> = present
<synsem local content psoa relation> = turn
<synsem local content psoa dir> =
    <synsem local category comps first synsem local content>
<synsem local content psoa turner> =
    <synsem local category subj first synsem local content>.

Word take:
verb-compn
<phon list first> = take
<synsem local category head form> = finite
<synsem local category subj first synsem local content index num> = plural
<synsem local content tense> = present
<synsem local content psoa relation> = turn
<synsem local content psoa dir> =
    <synsem local category comps first synsem local content>
<synsem local content psoa turner> =
    <synsem local category subj first synsem local content>.

% TAKE2 (as in "take Neil St.")

Word take:
verb-compn
<phon list first> = take
<synsem local category head form> = base
<synsem local content psoa relation> = takepath
<synsem local content psoa path> =
    <synsem local category comps first synsem local content>
<synsem local content psoa taker> =
    <synsem local category subj first synsem local content>.

Word take:
verb-compn
<phon list first> = take
<synsem local category head form> = finite
<synsem local category subj first synsem local content index per> =
    {first second}
<synsem local category subj first synsem local content index num> = singular
```

```
<synsem local content tense> = present
<synsem local content psoa relation> = takepath
<synsem local content psoa path> =
    <synsem local category comps first synsem local content>
<synsem local content psoa taker> =
    <synsem local category subj first synsem local content>.

Word take:
verb-compn
<phon list first> = take
<synsem local category head form> = finite
<synsem local category subj first synsem local content index num> = plural
<synsem local content tense> = present
<synsem local content psoa relation> = takepath
<synsem local content psoa path> =
    <synsem local category comps first synsem local content>
<synsem local content psoa taker> =
    <synsem local category subj first synsem local content>.

Word taken:
verb-compn
<phon list first> = taken
<synsem local category head form> = ppart
<synsem local content psoa relation> = takepicture
<synsem local content psoa picture> =
    <synsem local category comps first synsem local content>
<synsem local content psoa taker> =
    <synsem local category subj first synsem local content>.

Word turn:
verb-compadv
<phon list first> = turn
<synsem local category head form> = base
<synsem local content psoa relation> = turn
<synsem local content psoa dir> =
    <synsem local category comps first synsem local content>
<synsem local content psoa turner> =
    <synsem local category subj first synsem local content>.

Word turn:
verb-compadv
<phon list first> = turn
<synsem local category head form> = finite
<synsem local category subj first synsem local content index per> =
    {first second}
<synsem local category subj first synsem local content index num> = singular
<synsem local content tense> = present
<synsem local content psoa relation> = turn
<synsem local content psoa dir> =
    <synsem local category comps first synsem local content>
<synsem local content psoa turner> =
    <synsem local category subj first synsem local content>.

Word turn:
verb-compadv
<phon list first> = turn
<synsem local category head form> = finite
<synsem local category subj first synsem local content index num> = plural
<synsem local content tense> = present
<synsem local content psoa relation> = turn
<synsem local content psoa dir> =
    <synsem local category comps first synsem local content>
<synsem local content psoa turner> =
    <synsem local category subj first synsem local content>.

Word turned:
verb-compadj
<phon list first> = turned
<synsem local category head form> = {finite ppart}
<synsem local category comps first synsem local category subj first synsem> =
    <synsem local category head mod synsem>
<synsem local content tense> = past
<synsem local content psoa relation> = become
<synsem local content psoa quality> =
    <synsem local category comps first synsem local content restr rest first>
<synsem local content psoa becomer> =
    <synsem local category subj first synsem local content>.

Word turning:
verb-compadv
<phon list first> = turning
<synsem local category head form> = prog
<synsem local content psoa relation> = turn
<synsem local content psoa dir> =
    <synsem local category comps first synsem local content>
<synsem local content psoa turner> =
    <synsem local category subj first synsem local content>.
```

```
Word went:
verb-compadv
<phon list first> = went
<synsem local category head form> = finite
<synsem local content tense> = past
<synsem local content psoa relation> = go
<synsem local content psoa loc> =
    <synsem local category comps first synsem local content>
<synsem local content psoa goer> =
    <synsem local category subj first synsem local content>.

define bepredverb:
verb-compadj
<synsem local content psoa relation> = hasproperty
<synsem local content psoa property> =
    <synsem local category comps first synsem local content restr rest first>
<synsem local content psoa argument> =
    <synsem local category subj first synsem local content>.

Word be:
bepredverb
<phon list first> = be
<synsem local category head form> = base.

Word is:
bepredverb
<phon list first> = is
<synsem local category head form> = finite
<synsem local category subj first synsem local content index per> = third
<synsem local category subj first synsem local content index num> = singular
<synsem local content tense> = present.

Word are:
bepredverb
<phon list first> = are
<synsem local category head form> = finite
<synsem local category subj first synsem local content index per> = third
<synsem local category subj first synsem local content index num> = plural
<synsem local content tense> = present.

Word was:
bepredverb
<phon list first> = was
<synsem local category head form> = finite
```

```
<synsem local category subj first synsem local content index per> =
    {first third}
<synsem local category subj first synsem local content index num> = singular
<synsem local content tense> = past.

Word were:
bepredverb
<phon list first> = were
<synsem local category head form> = finite
<synsem local category subj first synsem local content index per> = third
<synsem local category subj first synsem local content index num> = plural
<synsem local content tense> = past.

Word appeared:
bepredverb
<phon list first> = appeared
<synsem local category head form> = finite
<synsem local content tense> = past.

define benomverb:
verb
<synsem local category comps first synsem local category head maj> = n
<synsem local category comps first synsem local category comps> = end
<synsem local category comps first synsem local category bar> = two
<synsem local category comps rest> = end
<synsem local content psoa relation> =
    <synsem local category comps first synsem local content>
<synsem local content psoa instance> =
    <synsem local category subj first synsem local content>.

Word is:
benomverb
<phon list first> = is
<synsem local category head form> = finite
<synsem local category subj first synsem local content index per> = third
<synsem local category subj first synsem local content index num> = singular
<synsem local content tense> = present.

define existentialbeverb:
subst
<synsem local category head maj> = v
<synsem local category comps first synsem local category head maj> = n
<synsem local category comps first synsem local category comps> = end
<synsem local category comps first synsem local category bar> = two
```

BEST COPY AVAILABLE

```
<synsem local category comps first synsem local content index per> = third
<synsem local category comps rest> = end
<synsem local category subj first synsem local category head maj> = n
<synsem local category subj first synsem local category head form> = there
<synsem local category subj first synsem local category bar> = zero
<synsem local category subj rest> = end
<synsem local content psoa relation> = exists
<synsem local content psoa instance> =
   <synsem local category comps first synsem local content>.

Word is:
existentialbeverb
<phon list first> = is
<synsem local category head form> = finite
<synsem local category comps first synsem local content index num> = singular
<synsem local content tense> = past.
```

Appendix B

Text Corpus Used For Grammar Development

```
%% These are the traffic accident reports.
%% This version gives the number of parses when semantics is consulted.

%% Four parses
par1-1 :- parse([num1,was,northbound,on,county,road,gravel,and,came,up,
onto,a,hill,and,num2,was,southbound,in,the,center,of,the,road]).

%% One parse
par1-2 :- parse([num1,swerved,to,the,left,to,avoid,hitting,num2,and,
flipped,over]).

%% Two parses
par2-1 :- parse([num1,was,southbound,on,elm]).

%% Two parses
par2-2 :- parse([num2,was,eastbound,on,elm]).

%% Eight parses
par2-3 :- parse([after,stopping,at,the,stopsign,num2,started,to,cross,the,
southbound,lanes,and,was,struck,in,the,left,side,by,num1]).

%% Theoretically can do this, but run out of memory
par3-1 :- parse([the,driver,of,num1,slowed,to,turn,left,on,elm,but,slid,on,
the,ice,covered,street,into,the,lane,to,her,right,in,the,front,of,num2]).
%% Two parses
par3-1-1 :- parse([the,driver,of,num1,slowed,to,turn,left,on,elm]).
```

```
%% Memory problem
par3-1-2 :- parse([the,driver,of,num1,slid,on,the,ice,covered,street,into,
the,lane,to,her,right,in,the,front,of,num2]).
%% Two parses
par3-1-2-1 :- parse([the,driver,of,num1,slid,on,the,ice,covered,street]).
%% Six parses
par3-1-2-2 :- parse([the,driver,of,num1,slid,on,the,ice,covered,street,into,
the,lane]).
%% Memory problem
par3-1-2-3 :- parse([the,driver,of,num1,slid,on,the,ice,covered,street,into,
the,lane,to,her,right]).

%% Theoretically can do this, but run into memory problems.
par3-2 :- parse([num2s,driver,steered,to,the,west,curb,went,over,the,curb,
hit,and,ran,over,a,stopsign,hit,a,stopsign,and,stopped,within,a,foot,of,a,
utility,pole]).
%% One parse
par3-2-1 :- parse([num2s,driver,steered,to,the,west,curb]).
%% One parse
par3-2-2 :- parse([num2s,driver,went,over,the,curb]).
%% One parse
par3-2-3 :- parse([num2s,driver,hit,a,stopsign]).
%% Two parses
par3-2-4 :- parse([num2s,driver,ran,over,a,stopsign]).
%% One parse
par3-2-5 :- parse([num2s,driver,stopped,within,a,foot,of,a,utility,pole]).

%% Four parses
par3-3 :- parse([num2s,left,front,fender,was,bent,against,the,tire,after,
hitting,num1s,right,rear]).

%% One parse
par3-4 :- parse([num1,went,on,across,elm,and,hit,a,utility,pole]).

%% One parse
par3-5 :- parse([no,injuries,were,reported]).

%% Four parses
par4-1 :- parse([num1,stopped,at,the,stopsign,after,leaving,ks,merchandise,
mart]).

%% Eight parses
par4-2 :- parse([the,driver,of,num1,pulled,out,to,make,a,left,turn,onto,
vermillion,street,to,go,south]).
```

```
%% Ten parses
par4-3 :- parse([the,driver,of,num2,was,northbound,on,vermillion,in,the,
left,lane,when,num1,pulled,out]).

%% One parse
par4-4 :- parse([num2,skidded,after,braking,and,hit,the,left,side,o>f,num1]).

%% One parse
par4-5 :- parse([the,driver,of,num2,and,a,passenger,were,injured]).

%% One parse
par4-6 :- parse([the,drivers,injuries,were,to,the,chest]).

%% Two parses
par4-7 :- parse([the,passenger,was,injured,in,the,lip,and,back]).

%% Six parses
par4-8 :- parse([the,front,seat,passengers,right,knee,and,neck,were,sore]).

%% Two parses
par4-9 :- parse([the,driver,of,num1,was,not,injured]).

%% Two parses
par4-10 :- parse([the,driver,of,num1,was,ticketed,for,failure,to,yield,
while,turning,left]).

%% Twelve parses
par5-1 :- parse([num1,was,eastbound,on,kirby,and,stated,that,she,saw,the,
light,go,to,yellow,and,was,trying,to,beat,it]).
%% Two parses
par5-1-1 :- parse([num1,was,eastbound,on,kirby]).
%% Two parses
par5-1-2 :- parse([num1,stated,that,she,saw,the,light,go,to,yellow]).
%% Two parses
par5-1-3 :- parse([num1,stated,that,she,was,trying,to,beat,it]).

%% Five parses
par5-2 :- parse([num2,was,westbound,on,kirby,in,the,lefthand,lane,
turning,left]).

%% One parse
par5-3 :- parse([num2,stated,that,both,he,and,another,vehicle,were,in,
the,turn,lane]).
```

```
%% One parse
par5-4 :- parse([the,other,vehicle,completed,its,turn,and,he,followed]).

%% Four parses
par5-5 :- parse([num2,stated,that,the,light,turned,red,while,he,was,in,
the,intersection]).

%% Six parses
par5-6 :- parse([he,was,clearing,the,intersection,when,the,accident,occurred]).

%% Two parses
par5-7 :- parse([witnesses,confirmed,that,the,light,was,red,when,num1,
entered,the,intersection]).

%% Memory problem.
par5-8 :- parse([num1s,momentum,after,the,accident,carried,her,through,
the,intersection,across,a,curb,and,into,a,sodded,grassy,area]).
%% Three parses
par5-8-1 :- parse([num1s,momentum,after,the,accident,carried,her,through,
the,intersection]).
%% Three parses
par5-8-2 :- parse([num1s,momentum,after,the,accident,carried,her,across,
a,curb]).
%% Three parses
par5-8-3 :- parse([num1s,momentum,after,the,accident,carried,her,into,a,
sodded,grassy,area]).

%% One parse
par5-9 :- parse([the,sod,was,damaged]).

%% Two parses
par5-10 :- parse([num1,was,ticketed,for,disobeying,a,traffic,control,signal]).

%% Six parses
par5-11 :- parse([all,injured,parties,were,transported,by,arrow,ambulance,
to,carle,hospital]).

%% One parse
par5-12 :- parse([photographs,of,the,accident,scene,were,taken,by,officer,
george,wissmiller]).

%% One parse
par5-13 :- parse([five,frames,were,taken]).
```

BEST COPY AVAILABLE

```
%% Sixteen parses
par6-1 :- parse([num1,was,going,north,on,duncan,road,at,approximately,
the,twelvehundred,block,and,hit,an,icy,spot]).

%% One parse
par6-2 :- parse([it,slid,west,to,num1s,left]).

%% Memory problem
par6-3 :- parse([the,car,spun,around,and,stopped,by,hitting,a,parked,car,
on,the,righthand,side,of,the,street]).
%% One parse
par6-3-1 :- parse([the,car,spun,around]).
%% Eleven parses
par6-3-2 :- parse([the,car,stopped,by,hitting,a,parked,car,on,the,
righthand,side,of,the,street]).
%% One parse
par6-3-2-1 :- parse([the,car,stopped,by,hitting,a,parked,car]).
%% Four parses
par6-3-2-2 :- parse([the,car,stopped,by,hitting,a,parked,car,on,the,
righthand,side]).

%% Memory problem
par7-1 :- parse([num2,was,northbound,on,fourth,street,approaching,
university,avenue]).
%% Two parses
par7-1-1 :- parse([num2,was,northbound,on,fourth,street]).

%% Memory problem
par7-2 :- parse([num1,was,westbound,from,the,alleyway,onto,fourth,street,
attempting,to,turn,left,onto,fourth,street]).
%% One parse
par7-2-1 :- parse([num1,was,westbound,from,the,alleyway]).
%% Six parses
par7-2-2 :- parse([num1,was,westbound,from,the,alleyway,onto,fourth,street]).
par7-2-3 :- parse([num1,was,westbound,from,the,alleyway,onto,fourth,street,
attempting,to,turn,left]).

%% Memory problem
par7-3 :- parse([num1,states,that,he,stopped,before,pulling,onto,fourth,
street,saw,no,oncoming,traffic,and,pulled,out,making,a,left,turn,onto,
fourth,street]).
%% Six parses
par7-3-1 :- parse([num1,states,that,he,stopped,before,pulling,onto,fourth,
street]).
%% One parse
par7-3-2 :- parse([num1,states,that,he,saw,no,oncoming,traffic]).
%% Twelve parses
par7-3-3 :- parse([num1,states,that,he,pulled,out,making,a,left,turn,onto,
fourth,street]).
%% One parse
par7-3-3-1 :- parse([num1,states,that,he,pulled,out]).
%% One parse
par7-3-3-2 :- parse([num1,states,that,he,pulled,out,making,a,left,turn]).

%% This should work, but runs into memory problems
par7-4 :- parse([num2,states,that,she,was,northbound,on,fourth,and,
suddenly,this,car,pulled,out,in,front,of,her,causing,her,to,collide,
with,the,vehicle,striking,the,drivers,side,door]).
%% Four parses
par7-4-1 :- parse([num2,states,that,she,was,northbound,on,fourth]).
%% This should work, but runs into memory problems
par7-4-2 :- parse([num2,states,that,suddenly,this,car,pulled,out,in,front,of,
her,causing,her,to,collide,with,the,vehicle,striking,the,drivers,side,door]).
%% Two parses
par7-4-2-1 :- parse([num2,states,that,suddenly,this,car,pulled,out]).
%% Four parses
par7-4-2-2 :- parse([num2,states,that,suddenly,this,car,pulled,out,in,front]).
%% Four parses
par7-4-2-3 :- parse([num2,states,that,suddenly,this,car,pulled,out,in,front,
of,her]).

%% Six parses
par8-1 :- parse([num1,and,num2,were,eastbound,on,i74]).

%% One parse
par8-2 :- parse([the,road,was,snow,covered,and,icy]).

%% One parse
par8-3 :- parse([num1,hit,a,slick,spot,and,lost,control]).

%% One parse
par8-4 :- parse([num1,slid,sideways,into,the,passing,lane]).

%% Four parses
par8-5 :- parse([num2,saw,num1,in,his,lane,but,was,unable,to,get,stopped,
before,striking,num1]).
```

```
%% Nine parses
par9-1 :- parse([according,to,the,driver,of,num2,the,accident,was,her,fault]).

%% Two parses
par9-2 :- parse([num2,was,westbound,on,university,at,fourth]).

%% One parse
par9-3 :- parse([the,driver,of,num2,had,just,left,a,local,hospital]).

%% Two parses
par9-4 :- parse([she,did,not,realize,the,signal,light,had,turned,red]).

%% Two parses
par9-5 :- parse([num2,entered,the,intersection,and,struck,num1,southbound,
on,fourth]).

%% This runs into memory problems
par10a-1 :- parse([num1,was,southbound,on,the,frontage,road,behind,the,ford,
dealership,worden,martin,on,the,hill,approximately,thirty,feet,from,the,
intersection,where,it,meets,devonshire,road]).
%% One parse
par10a-1-1 :- parse([num1,was,southbound,on,the,frontage,road]).
%% Thirteen parses
par10a-1-2 :- parse([num1,was,southbound,on,the,frontage,road,behind,the,
ford,dealership,worden,martin]).
%% Twenty-three parses
par10a-1-3 :- parse([num1,was,southbound,on,the,frontage,road,behind,the,
ford,dealership,worden,martin,on,the,hill]).
%% Memory problems.
par10a-1-4 :- parse([num1,was,southbound,on,the,frontage,road,behind,the,
ford,dealership,worden,martin,on,the,hill,approximately,thirty,feet,from,
the,intersection]).

%% Three parses
par10a-2 :- parse([there,was,a,solid,sheet,of,ice]).

%% Memory problems
par10a-3 :- parse([the,car,slid,up,over,the,hill,and,into,the,intersection,
hitting,num2,who,was,going,west,on,devonshire]).
%% One parse
par10a-3-1 :- parse([the,car,slid,up,over,the,hill]).
%% This should work, but runs into memory problems.
par10a-3-2 :- parse([the,car,slid,into,the,intersection,hitting,num2,who,
was,going,west,on,devonshire]).
```

BEST COPY AVAILABLE

```
%% One parse
par10a-3-2-1 :- parse([the,car,slid,into,the,intersection]).
%% Two parses
par10a-3-2-2 :- parse([the,car,slid,into,the,intersection,hitting,num2]).
%% Six parses
par10a-3-2-3 :- parse([the,car,slid,into,the,intersection,hitting,num2,who,
was,going,west]).

%% This causes memory overflow.
par10b-1 :- parse([i,was,driving,a,cadillac,twenty,mph,west,on,devonshire,
when,i,saw,the,buick,heading,at,the,intersection,without,slowing]).
%% One parse
par10b-1-1 :- parse([i,was,driving,a,cadillac]).
%% Two parses
par10b-1-2 :- parse([i,was,driving,a,cadillac,twenty,mph]).
%% Twenty-eight parses
par10b-1-3 :- parse([i,was,driving,a,cadillac,twenty,mph,west,on,devonshire]).

%% Forty-five parses
par10b-2 :- parse([unable,to,dodge,in,time,i,arrived,at,the,intersection,
and,was,hit,by,the,same,buick,in,the,front,right,side]).
%% Three parses
par10b-2-1 :- parse([unable,to,dodge,in,time,i,arrived,at,the,intersection]).
%% Sixteen parses
par10b-2-2 :- parse([i,was,hit,by,the,same,buick,in,the,front,right,side]).

%% Memory problem.
par11a-1 :- parse([the,insured,came,out,of,a,building,to,get,into,his,vehicle,
and,the,owner,of,num2,stated,that,meuser,had,backed,into,his,vehicle]).
%% One parse
par11a-1-1 :- parse([the,insured,came,out,of,a,building,to,get,into,his,
vehicle]).
%% One parse
par11a-1-2 :- parse([the,owner,of,num2,stated,that,meuser,had,backed,into,
his,vehicle]).

%% One parse
par11a-2 :- parse([meuser,stated,that,he,did,not,hit,num2]).

%% Ten parses
par11a-3 :- parse([num2,had,previous,damage,and,had,rust,on,it,where,
supposedly,meuser,had,hit,num2]).
%% One parse
par11a-3-1 :- parse([num2,had,previous,damage]).
```

```
%% Four parses
par11a-3-2 :- parse([num2,had,rust,on,it]).
%% One parse
par11a-3-3 :- parse([meuser,had,hit,num2]).
%% One parse
par11a-3-4 :- parse([supposedly,meuser,had,hit,num2]).

%% Two parses
par11b-1 :- parse([i,spoke,to,the,insured,paul,a,meuser]).

%% Sixty-seven parses
par11b-2 :- parse([he,was,at,a,garage,sale,in,tavernier,parallel,parked,
in,the,street]).

%% Four parses
par11b-3 :- parse([when,he,was,leaving,he,had,to,back,up,a,little,to,exit,
his,space]).

%% This runs into memory problems
par11b-4 :- parse([then,he,started,forward,and,the,claimant,came,running,
up,to,the,insureds,truck,yelling,that,the,insured,had,backed,into,the,
claimants,car,which,was,parked,behind,the,insured]).
%% Two parses
par11b-4-1 :- parse([then,he,started,forward]).
%% This runs into memory problems
par11b-4-2 :- parse([the,claimant,came,running,up,to,the,insureds,truck,
yelling,that,the,insured,had,backed,into,the,claimants,car,which,was,
parked,behind,the,insured]).
par11b-4-2-1 :- parse([the,claimant,came,running,up,to,the,insureds,truck]).
%% Eight parses
par11b-4-2-2 :- parse([the,insured,had,backed,into,the,claimants,car,which,
was,parked,behind,the,insured]).
%% Two parses
par11b-4-2-2-1 :- parse([the,insured,had,backed,into,the,claimants,car]).
%% Two parses
par11b-4-2-2-2 :- parse([the,claimants,car,was,parked,behind,the,insured]).

%% Seven parses
par11b-5 :- parse([the,insured,inspected,the,claimants,vehicle,and,saw,
damage,to,its,grill,that,appeared,very,old,and,rusted]).
%% Four parses
par11b-5-1 :- parse([the,insured,inspected,the,claimants,vehicle,and,saw,
damage,to,its,grill]).
%% One parse
```

```
par11b-5-2 :- parse([the,damage,appeared,very,old,and,rusted]).

%% One parse
par11b-6 :- parse([also,the,insured,and,his,wife,did,not,hear,or,feel,
any,impact]).

%% Thirty-seven parses
par11b-7 :- parse([he,said,that,the,claimant,was,very,belligerent,to,the,
point,that,the,insured,just,gave,the,claimant,his,insurance,information,
because,he,was,afraid,he,d,lose,his,temper,with,the,claimant]).
%% Two parses
par11b-7-1 :- parse([he,said,that,the,claimant,was,very,belligerent,to,the,
point,that,the,insured,just,gave,the,claimant,his,insurance,information]).
%% Two parses
par11b-7-1-1 :- parse([he,said,that,the,claimant,was,very,belligerent]).
%% One parse
par11b-7-1-2 :- parse([the,insured,just,gave,the,claimant,his,insurance,
information]).
%% Nine parses
par11b-7-2 :- parse([the,insured,just,gave,the,claimant,his,insurance,
information,because,he,was,afraid,he,d,lose,his,temper,with,the,claimant]).
%% Three parses
par11b-7-2-1 :- parse([he,was,afraid,he,d,lose,his,temper,with,the,claimant]).

%% One parse
par11b-8 :- parse([the,insured,still,does,not,think,he,hit,the,claimants,
vehicle]).

%% One parse
par11b-9 :- parse([there,were,no,disinterested,witnesses,as,far,as,the,
insured,knows]).

%% Memory problem
par12a-1 :- parse([num1,was,eastbound,on,university,avenue,turning,left,
onto,cunningham,avenue,northbound,in,the,outside,turn,lane]).
%% Two parses
par12a-1-1 :- parse([num1,was,eastbound,on,university,avenue]).
%% Six parses
par12a-1-2 :- parse([num1,was,eastbound,on,university,avenue,turning,left]).
%% One parse
par12a-1-3 :- parse([num1,was,eastbound,on,university,avenue,turning,left,
onto,cunningham,avenue]).
%% Fourteen parses
par12a-1-4 :- parse([num1,was,eastbound,on,university,avenue,turning,left,
onto,cunningham,avenue,northbound]).

%% Memory problem
par12a-2 :- parse([num2,was,northbound,on,vine,street,approaching,
university,avenue]).

%% One parse
par12a-3 :- parse([the,driver,of,num1,was,following,other,vehicles,and,
entered,the,intersection,after,the,traffic,signal,showed,red]).

%% Eight parses
par12a-4 :- parse([the,driver,of,num2,proceeded,forward,and,num1,
collided,with,the,left,rear,corner,of,num2]).
%% One parse
par12a-4-1 :- parse([the,driver,of,num2,proceeded,forward]).
%% Two parses
par12a-4-2 :- parse([num1,collided,with,the,left,rear,corner,of,num2]).

%% Two parses
par12a-5 :- parse([two,child,passengers,in,num1,were,slightly,injured]).

%% One parse
par12a-6 :- parse([both,vehicles,sustained,moderate,damage]).

%% Ten parses
par12b-1 :- parse([num1,was,eastbound,on,university,avenue,and,num2,was,
northbound,on,cunningham,avenue,in,urbana]).
%% Two parses
par12b-1-1 :- parse([num1,was,eastbound,on,university,avenue]).
%% Five parses
par12b-1-2 :- parse([num2,was,northbound,on,cunningham,avenue,in,urbana]).

%% Can't do rel clause
par12b-2 :- parse([num1,went,through,the,intersection,on,a,yellow,light,
that,apparently,had,just,turned,red]).
%% One parse
par12b-2-1 :- parse([num1,went,through,the,intersection]).
%% Three parses
par12b-2-2 :- parse([num1,went,through,the,intersection,on,a,yellow,light]).

%% One parse
par12b-3 :- parse([num2,had,a,green,light,while,going,through,that,same,
intersection]).
```

```
%% Four parses
par12b-4 :- parse([num1,struck,num2,on,the,left,rear,side]).

%% One parse
par13-1 :- parse([num2,was,going,down,the,road]).

%% One parse
par13-2 :- parse([num1,pulled,from,a,stopsign,and,didnt,see,num2]).

%% One parse
par13-3 :- parse([cars,were,parked,on,each,side,of,the,stopsign]).

%% One parse
par13-4 :- parse([num1,pulled,out]).

%% One parse
par13-5 :- parse([a,collision,occurred]).

%% Three parses
par13-6 :- parse([num2,admitted,to,num1,that,he,was,speeding]).

%% Can't do the final "I said"
par14-1 :- parse([a,semitrailer,truck,rammed,through,a,posted,
construction,zone,on,interstate57,wednesday,and,hit,an,asphalt,
grinder,killing,three,people,police,said]).
%% Seventy-five parses
par14-1-1 :- parse([a,semitrailer,truck,rammed,through,a,posted,
construction,zone,on,interstate57,wednesday,and,hit,an,asphalt,
grinder,killing,three,people]).
%% Five parses
par14-1-1-1 :- parse([a,semitrailer,truck,rammed,through,a,posted,
construction,zone,on,interstate57,wednesday]).
%% Two parses
par14-1-1-2 :- parse([a,semitrailer,truck,hit,an,asphalt,grinder,
killing,three,people]).

%% Can't do final s-modifying adv-phrase
par14-2 :- parse([the,semi,carrying,a,load,of,aluminum,cans,caught,fire,
and,was,destroyed,according,to,state,police]).
%% Seven parses
par14-2-1 :- parse([the,semi,carrying,a,load,of,aluminum,cans,caught,fire,
and,was,destroyed]).

%% Can't do the final "I said"
```

```
par14-3 :- parse([the,trucks,driver,and,a,passenger,and,the,operator,of,
the,grinder,were,killed,police,said]).
%% Two parses
par14-3-1 :- parse([the,trucks,driver,and,a,passenger,and,the,operator,of,
the,grinder,were,killed]).

%% Can't do the final "X said"
par14-4 :- parse([it,was,not,known,if,the,trucks,occupants,died,from,the,
impact,or,the,fire,troopers,said]).
par14-4-1 :- parse([it,was,not,known,if,the,trucks,occupants,died,from,
the,impact,or,the,fire]).

%% Twenty-seven parses
par14-5 :- parse([the,identities,of,the,victims,in,the,truck,were,not,
released,by,early,evening]).

%% Can't do the final "X said"
par14-6 :- parse([the,operator,of,the,grinder,edward,ladewick,of,beecher,
illinois,was,pronounced,dead,at,saint,anthony,hospital,in,effingham,state,
police,trooper,charles,kline,said]).
%% Twenty-eight parses
par14-6-1 :- parse([the,operator,of,the,grinder,edward,ladewick,of,beecher,
illinois,was,pronounced,dead,at,saint,anthony,hospital,in,effingham]).

%% This runs into memory problems
par14-7 :- parse([police,said,the,southbound,truck,failed,to,merge,into,
the,right,lane,in,its,approach,to,the,construction,zone,and,struck,the,
grinder,owned,by,howell,asphalt]).
%% Thirty-three parses
par14-7-1 :- parse([police,said,the,southbound,truck,failed,to,merge,into,
the,right,lane,in,its,approach,to,the,construction,zone]).
%% Two parses
par14-7-2 :- parse([police,said,the,southbound,truck,struck,the,grinder,
owned,by,howell,asphalt]).

%% Three parses
par14-8 :- parse([southbound,traffic,was,rerouted,through,neoga,and,
effingham,during,the,morning,and,early,afternoon,until,the,accident,
scene,could,be,cleared]).

%% One parse
par15-1 :- parse([i,was,in,num1,and,came,to,a,halt,behind,num3]).

%% Six parses
```

```
par15-2 :- parse([we,had,stopped,as,the,tail,end,of,a,halted,line,of,cars]).

%% Five parses
par15-3 :- parse([i,reached,a,complete,stop,about,six,to,eight,feet,
behind,num3,which,was,just,up,to,fifth,street]).
%% One parse
par15-3-1 :- parse([i,reached,a,complete,stop]).
%% Three parses
par15-3-2 :- parse([i,reached,a,complete,stop,behind,num3]).
par15-3-3 :- parse([i,reached,a,complete,stop,behind,num3,which,was,
just,up,to,fifth,street]).

%% Fifty-three parses
par15-4 :- parse([simultaneous,with,completing,my,stop,i,heard,screeching,
tires,and,looked,in,my,rearview,mirror,in,time,to,see,num2,approaching]).
%% One parse
par15-4-1 :- parse([simultaneous,with,completing,my,stop,i,heard,
screeching,tires]).
%% Three parses
par15-4-2 :- parse([i,looked,in,my,rearview,mirror,in,time,to,see,num2,
approaching]).
%% One parse
par15-4-2-1 :- parse([i,looked,in,my,rearview,mirror]).
%% Three parses
par15-4-2-2 :- parse([i,looked,in,my,rearview,mirror,in,time]).

%% One parse
par15-5 :- parse([i,had,just,enough,time,to,realize,that,he,was,going,
to,impact,and,to,brace,myself]).

%% Three parses
par15-6 :- parse([num2,collided,with,me,num1,and,both,vehicles,continued,
forward,and,then,impacted,num3]).
%% Two parses
par15-6-1 :- parse([num2,collided,with,me,num1]).
%% One parse
par15-6-2 :- parse([both,vehicles,continued,forward,and,then,impacted,num3]).
%% One parse
par15-6-2-1 :- parse([both,vehicles,continued,forward]).
%% One parse
par15-6-2-2 :- parse([both,vehicles,then,impacted,num3]).

%% Memory problems.
par15-7 :- parse([the,first,impact,severely,reshaped,num2s,front,bumper,
grill,and,the,rear,onethird,of,num1]).
%% One parse
par15-7-1 :- parse([the,first,impact,severely,reshaped,num2s,front,bumper]).
%% One parse
par15-7-2 :- parse([the,first,impact,severely,reshaped,the,rear,onethird,
of,num1]).

%% Can't do sentences joined by a comma
par15-8 :- parse([the,second,impact,further,compressed,num1,and,damaged,
num3,also,it,injured,the,frontseat,passenger,of,num3,and,caused,me,to,
strike,my,nose,breaking,it]).
%% One parse
par15-8-1 :- parse([the,second,impact,further,compressed,num1,and,
damaged,num3]).
%% This runs into memory problems
par15-8-2 :- parse([also,it,injured,the,frontseat,passenger,of,num3,
and,caused,me,to,strike,my,nose,breaking,it]).
%% One parse
par15-8-2-1 :- parse([also,it,injured,the,frontseat,passenger,of,num3]).
%% Twelve parses
par15-8-2-2 :- parse([also,it,caused,me,to,strike,my,nose,breaking,it]).
```

Appendix C

Real-World Knowledge Base

```
%% This file contains the real-world knowledge used by the system in
%% parsing.  For each type of world knowledge, only a few
%% representative examples are given.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% Knowledge of the world
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

C.1 Slot-Filler Information

C.1.1 Relation-Role Information

```
%%* Defaults -- e.g. for pronouns

%% can_be_role_of(nil,_,_).
%% can_be_role_of(_,_,nil).

%% Cases where something's semantics is the same as that of its
%% complement; e.g. auxiliary verbs.

can_be_role_of(X,same,X).


%%* Predicate adjectives

can_be_role_of(Arg,property,hasproperty) :-
```

```
        is_property(Arg).
can_be_role_of(_,argument,hasproperty).


%%* Adjectives with complements

can_be_role_of(Arg,feared,afraid) :-
        is_proposition(Arg).

can_be_role_of(Arg,unablee,unable) :-
        is_action(Arg).


%%* Complements of prepositions.

can_be_role_of(Arg,comparator,accordingto) :-
        is_person(Arg).

can_be_role_of(Arg,comparator,across) :-
        is_street(Arg).
can_be_role_of(curb,comparator,across).

can_be_role_of(Arg,comparator,against) :-
        is_phys_obj(Arg).

can_be_role_of(Arg,comparator,along) :-
        is_street(Arg).

can_be_role_of(Arg,comparator,as) :-
        is_position(Arg).

can_be_role_of(intersection,comparator,at).
can_be_role_of(Arg,comparator,at) :-
        is_building(Arg).
can_be_role_of(Arg,comparator,at) :-
        is_phys_obj(Arg).
can_be_role_of(Arg,comparator,at) :-
        is_street(Arg).

can_be_role_of(Arg,comparator,before) :-
        is_event(Arg).

can_be_role_of(Arg,comparator,behind) :-
        is_building(Arg).
```

BEST COPY AVAILABLE

```
can_be_role_of(Arg,comparator,behind) :-
        is_person(Arg).
can_be_role_of(Arg,comparator,behind) :-
        is_vehicle(Arg).

can_be_role_of(Arg,comparator,by) :-
        is_building(Arg).
can_be_role_of(Arg,comparator,by) :-
        is_company(Arg).
can_be_role_of(Arg,argument,by) :-
        is_person(Arg).
can_be_role_of(Arg,argument,by) :-
        is_vehicle(Arg).

can_be_role_of(Arg,comparator,dest) :-
        is_body_part(Arg).
can_be_role_of(Arg,comparator,dest) :-
        is_building(Arg).
can_be_role_of(Arg,comparator,dest) :-
        is_phys_obj(Arg).
can_be_role_of(halt,comparator,dest).

can_be_role_of(Arg,comparator,dir) :-
        is_direction(Arg).

can_be_role_of(Arg,comparator,down) :-
        is_street(Arg).

can_be_role_of(Arg,comparator,for) :-
        is_event(Arg).

can_be_role_of(Arg,comparator,loc) :-
        is_direction(Arg).
can_be_role_of(Arg,comparator,loc) :-
        is_phys_obj(Arg).

can_be_role_of(Arg,argument,means) :-
        is_action(Arg).

can_be_role_of(Arg,comparator,outof) :-
        is_building(Arg).

can_be_role_of(Arg,comparator,over) :-
        is_phys_obj(Arg).
```

```
can_be_role_of(Arg,comparator,path_change) :-
        is_street(Arg).

can_be_role_of(Arg,comparator,partitive) :-
        is_object(Arg).

can_be_role_of(Arg,comparator,past) :-
        is_building(Arg).

can_be_role_of(Arg,comparator,simultaneouswith) :-
        is_event(Arg).

can_be_role_of(intersection,comparator,through).
can_be_role_of(Arg,comparator,through) :-
        is_area(Arg).
can_be_role_of(Arg,comparator,through) :-
        is_town(Arg).

can_be_role_of(Arg,comparator,while) :-
        is_event(Arg).

can_be_role_of(Arg,comparator,with) :-
        is_person(Arg).
can_be_role_of(Arg,comparator,with) :-
        is_phys_object(Arg).

can_be_role_of(Arg,comparator,within) :-
        is_measure(Arg).


%%* Complements of subordinating conjunctions

can_be_role_of(Arg,subevent,after) :-
        is_event(Arg).

can_be_role_of(Arg,subevent,while) :-
        is_event(Arg).

can_be_role_of(Arg,subevent,when) :-
        is_event(Arg).


%%* Verbal relations
```

```
can_be_role_of(Arg,admitter,admit) :-
        is_person(Arg).
can_be_role_of(Arg,admitter,admit) :-
        is_vehicle(Arg).
can_be_role_of(Arg,hearer,admit) :-
        is_person(Arg).
can_be_role_of(Arg,hearer,admit) :-
        is_vehicle(Arg).
can_be_role_of(Arg,admitted,admit) :-
        is_proposition(Arg).

can_be_role_of(Arg,approacher,approach) :-
        is_person(Arg).
can_be_role_of(Arg,approacher,approach) :-
        is_vehicle(Arg).
can_be_role_of(Arg,approached,approach) :-
        is_area(Arg).
can_be_role_of(Arg,approached,approach) :-
        is_street(Arg).

can_be_role_of(Arg,arriver,arrive) :-
        is_person(Arg).
can_be_role_of(Arg,arriver,arrive) :-
        is_vehicle(Arg).
can_be_role_of(at,loc,arrive).

can_be_role_of(Arg,bearer,bear) :-
        is_person(Arg).
can_be_role_of(Arg,bearer,bear) :-
        is_vehicle(Arg).
can_be_role_of(Arg,loc,bear) :-
        is_direction(Arg).
can_be_role_of(dir,loc,bear).

can_be_role_of(Arg,quality,become) :-
        is_property(Arg).
can_be_role_of(Arg,becomer,become) :-
        is_phys_obj(Arg).
can_be_role_of(Arg,becomer,become) :-
        is_person(Arg).

can_be_role_of(Arg,bender,bend) :-
        is_phys_obj(Arg).
```

```
can_be_role_of(Arg,bendee,bend) :-
        is_phys_obj(Arg).

can_be_role_of(Arg,braker,brake) :-
        is_vehicle(Arg).

can_be_role_of(Arg,breaker,break) :-
        is_person(Arg).
can_be_role_of(Arg,breakee,break) :-
        is_body_part(Arg).

can_be_role_of(momentum,carrier,carry).
can_be_role_of(Arg,carrier,carry) :-
        is_vehicle(Arg).
can_be_role_of(Arg,carried,carry) :-
        is_person(Arg).
can_be_role_of(Arg,carried,carry) :-
        is_phys_obj(Arg).
can_be_role_of(through,loc,carry).

can_be_role_of(Arg,catcher,catchfire) :-
        is_phys_obj(Arg).
can_be_role_of(fire,caught,catchfire).

can_be_role_of(Arg,causer,cause) :-
        is_event(Arg).
can_be_role_of(Arg,causee,cause) :-
        is_person(Arg).
can_be_role_of(Arg,causedevent,cause) :-
        is_event(Arg).

can_be_role_of(Arg,clearer,clear) :-
        is_person(Arg).
can_be_role_of(intersection,clearee,clear).

can_be_role_of(Arg,comer,come) :-
        is_person(Arg).
can_be_role_of(Arg,comer,come) :-
        is_vehicle(Arg).
can_be_role_of(Arg,loc,come) :-
        is_change_of_loc(Arg).

can_be_role_of(Arg,confirmer,confirm) :-
        is_person(Arg).
```

```
can_be_role_of(Arg,confirmed,confirm) :-
        is_proposition(Arg).

can_be_role_of(Arg,continuer,continue) :-
        is_person(Arg).
can_be_role_of(Arg,continuer,continue) :-
        is_vehicle(Arg).
can_be_role_of(Arg,loc,continue) :-
        is_direction(Arg).
can_be_role_of(dir,loc,continue).

can_be_role_of(Arg,crosser,cross) :-
        is_person(Arg).
can_be_role_of(Arg,crosser,cross) :-
        is_street(Arg).
can_be_role_of(intersection,crossee,cross).

can_be_role_of(Arg,disobeyer,disobey) :-
        is_person(Arg).
can_be_role_of(Arg,disobeyee,disobey) :-
        is_tcd(Arg).

can_be_role_of(Arg,dodger,dodge) :-
        is_person(Arg).

can_be_role_of(Arg,driver,drive) :-
        is_person(Arg).
can_be_role_of(Arg,drivee,drive) :-
        is_vehicle(Arg).

can_be_role_of(Arg,enterer,enter) :-
        is_person(Arg).
can_be_role_of(Arg,enterer,enter) :-
        is_vehicle(Arg).
can_be_role_of(intersection,enteree,enter).

can_be_role_of(Arg,instance,exists) :-
        is_phys_obj(Arg).
can_be_role_of(Arg,instance,exists) :-
        is_event(Arg).

can_be_role_of(Arg,flipper,flip) :-
        is_vehicle(Arg).
can_be_role_of(over,loc,flip).
can_be_role_of(Arg,giver,give) :-
        is_person(Arg).
can_be_role_of(Arg,recipient,give) :-
        is_person(Arg).
can_be_role_of(Arg,object,give) :-
        is_phys_obj(Arg).

can_be_role_of(Arg,goer,go) :-
        is_person(Arg).
can_be_role_of(Arg,goer,go) :-
        is_vehicle(Arg).
can_be_role_of(light,goer,go).
can_be_role_of(Arg,loc,go) :-
        is_direction(Arg).
can_be_role_of(dir,loc,go).

can_be_role_of(Arg,halter,halt) :-
        is_person(Arg).
can_be_role_of(Arg,halted,halt) :-
        is_vehicle(Arg).

can_be_role_of(Arg,header,head) :-
        is_person(Arg).
can_be_role_of(Arg,header,head) :-
        is_vehicle(Arg).
can_be_role_of(Arg,loc,head) :-
        is_direction(Arg).
can_be_role_of(Arg,loc,head) :-
        is_change_of_loc(Arg).

can_be_role_of(Arg,hitter,hit) :-
        is_person(Arg).
can_be_role_of(Arg,hitter,hit) :-
        is_vehicle(Arg).
can_be_role_of(Arg,hittee,hit) :-
        is_vehicle(Arg).
can_be_role_of(Arg,hittee,hit) :-
        is_phys_obj(Arg).
can_be_role_of(spot,hittee,hit).

can_be_role_of(impact,injurer,injure).
can_be_role_of(Arg,injurer,injure) :-
        is_vehicle(Arg).
```

```
can_be_role_of(Arg,injured,injure) :-
        is_person(Arg).

can_be_role_of(Arg,inspecter,inspect) :-
        is_person(Arg).
can_be_role_of(Arg,inspectee,inspect) :-
        is_vehicle(Arg).

can_be_role_of(Arg,killer,kill) :-
        is_person(Arg).
can_be_role_of(Arg,killer,kill) :-
        is_vehicle(Arg).
can_be_role_of(Arg,killee,kill) :-
        is_person(Arg).

can_be_role_of(Arg,knower,know) :-
        is_person(Arg).

can_be_role_of(Arg,looker,look) :-
        is_person(Arg).
can_be_role_of(Arg,loc,look) :-
        is_direction(Arg).

can_be_role_of(Arg,loser,lose) :-
        is_person(Arg).
can_be_role_of(Arg,loser,lose) :-
        is_vehicle(Arg).
can_be_role_of(control,losee,lose).
can_be_role_of(temper,losee,lose).

can_be_role_of(Arg,meeter,meet) :-
        is_street(Arg).
can_be_role_of(Arg,meetee,meet) :-
        is_street(Arg).

can_be_role_of(Arg,merger,merge) :-
        is_vehicle(Arg).
can_be_role_of(lane,loc,merge).

can_be_role_of(Arg,event,occur) :-
        is_event(Arg).

can_be_role_of(Arg,owner,own) :-
        is_company(Arg).
```

```
can_be_role_of(Arg,owner,own) :-
        is_person(Arg).
can_be_role_of(Arg,owned,own) :-
        is_phys_obj(Arg).

can_be_role_of(Arg,parker,park) :-
        is_person(Arg).
can_be_role_of(Arg,parked,park) :-
        is_vehicle(Arg).

can_be_role_of(Arg,proceeder,proceed) :-
        is_person(Arg).
can_be_role_of(Arg,proceeder,proceed) :-
        is_vehicle(Arg).
can_be_role_of(forward,loc,proceed).

can_be_role_of(Arg,pronouncer,pronounce) :-
        is_person(Arg).
can_be_role_of(Arg,pronounced,pronounce) :-
        is_person(Arg).

can_be_role_of(Arg,puller,pull) :-
        is_person(Arg).
can_be_role_of(Arg,puller,pull) :-
        is_vehicle(Arg).
can_be_role_of(Arg,loc,pull) :-
        is_direction(Arg).

can_be_role_of(Arg,rammer,ram) :-
        is_person(Arg).
can_be_role_of(Arg,rammer,ram) :-
        is_vehicle(Arg).
can_be_role_of(Arg,loc,ram) :-
        is_change_of_loc(Arg).

can_be_role_of(Arg,runner,run) :-
        is_person(Arg).
can_be_role_of(Arg,runner,run) :-
        is_vehicle(Arg).
can_be_role_of(Arg,loc,run) :-
        is_direction(Arg).

can_be_role_of(Arg,runnerover,runover) :-
        is_person(Arg).
```

```
can_be_role_of(Arg,runnerover,runover) :-
        is_vehicle(Arg).
can_be_role_of(Arg,runover,runover) :-
        is_phys_obj(Arg).

can_be_role_of(Arg,realizer,realize) :-
        is_person(Arg).
can_be_role_of(Arg,realized,realize) :-
        is_proposition(Arg).

can_be_role_of(Arg,releaser,release) :-
        is_person(Arg).
can_be_role_of(Arg,released,release) :-
        is_information(Arg).

can_be_role_of(Arg,rerouter,reroute) :-
        is_person(Arg).
can_be_role_of(traffic,rerouted,reroute).

can_be_role_of(Arg,sayer,say) :-
        is_person(Arg).
can_be_role_of(Arg,said,say) :-
        is_proposition(Arg).

can_be_role_of(Arg,shower,show) :-
        is_phys_obj(Arg).
can_be_role_of(Arg,shown,show) :-
        is_property(Arg).

can_be_role_of(Arg,skidder,skid) :-
        is_vehicle(Arg).

can_be_role_of(Arg,slider,slide) :-
        is_person(Arg).
can_be_role_of(Arg,slider,slide) :-
        is_vehicle(Arg).
can_be_role_of(Arg,dir,slide) :-
        is_direction(Arg).

can_be_role_of(Arg,speaker,speak) :-
        is_person(Arg).
can_be_role_of(Arg,addressee,speak) :-
        is_person(Arg).
```

```
can_be_role_of(Arg,spinner,spin) :-
        is_vehicle(Arg).
can_be_role_of(Arg,loc,spin) :-
        is_direction(Arg).

can_be_role_of(Arg,steerer,steer) :-
        is_person(Arg).
can_be_role_of(Arg,steerer,steer) :-
        is_vehicle(Arg).
can_be_role_of(Arg,dir,steer) :-
        is_direction(Arg).
can_be_role_of(dest,dest,steer).
can_be_role_of(dir,dir,steer).

can_be_role_of(Arg,stopper,stop) :-
        is_person(Arg).
can_be_role_of(Arg,stopper,stop) :-
        is_vehicle(Arg).
can_be_role_of(Arg,stopped,stop) :-
        is_action(Arg).
can_be_role_of(at,loc,stop).

can_be_role_of(Arg,sustainer,sustain) :-
        is_vehicle(Arg).
can_be_role_of(damage,sustained,sustain).

can_be_role_of(Arg,swerver,swerve) :-
        is_person(Arg).
can_be_role_of(Arg,swerver,swerve) :-
        is_vehicle(Arg).
can_be_role_of(Arg,dir,swerve) :-
        ir_direction(Arg).
can_be_role_of(Arg,dir,swerve) :-
        is_change_of_loc(Arg).

can_be_role_of(Arg,taker,takepath) :-
        is_person(Arg).
can_be_role_of(Arg,path,takepath) :-
        is_street(Arg).

can_be_role_of(Arg,taker,takepicture) :-
        is_person(Arg).
can_be_role_of(frame,picture,takepicture).
can_be_role_of(photograph,picture,takepicture).
can_be_role_of(Arg,thinker,think) :-
        is_person(Arg).
can_be_role_of(Arg,thought,think) :-
        is_proposition(Arg).

can_be_role_of(Arg,ticketer,ticket) :-
        is_person(Arg).
can_be_role_of(Arg,ticketee,ticket) :-
        is_person(Arg).
can_be_role_of(Arg,ticketee,ticket) :-
        is_vehicle(Arg).

can_be_role_of(Arg,turner,turn) :-
        is_person(Arg).
can_be_role_of(Arg,turner,turn) :-
        is_vehicle(Arg).
can_be_role_of(Arg,dir,turn) :-
        is_direction(Arg).
can_be_role_of(Arg,dir,turn) :-
        is_change_of_loc(Arg).

can_be_role_of(Arg,trier,try) :-
        is_person(Arg).
can_be_role_of(Arg,trier,try) :-
        is_vehicle(Arg).
can_be_role_of(Arg,tried,try) :-
        is_action(Arg).

can_be_role_of(Arg,yeller,yell) :-
        is_person(Arg).
can_be_role_of(Arg,yeller,yell) :-
        is_vehicle(Arg).
can_be_role_of(Arg,yelled,yell) :-
        is_proposition(Arg).

can_be_role_of(Arg,yielder,yield) :-
        is_person(Arg).
can_be_role_of(Arg,yielder,yield) :-
        is_vehicle(Arg).
```

C.1.2 Modifier-Modified Information

```
%%* Defaults -- e.g. pronouns

%% can_modify(nil,_).
%% can_modify(_,nil).

%%* Adjectives

can_modify(about,Arg) :- is_measure(Arg).
can_modify(approximately,Arg) :- is_measure(Arg).

can_modify(belligerent,Arg) :- is_person(Arg).

can_modify(complete,Arg) :- is_maneuver(Arg).

can_modify(dead,Arg) :- is_person(Arg).

can_modify(disinterested,Arg) :- is_person(Arg).

can_modify(early,Arg) :- is_time_chunk(Arg).

can_modify(enough,Arg) :- is_commodity(Arg).

can_modify(icy,spot).
can_modify(icy,Arg) :-
        is_street(Arg).

can_modify(local,Arg) :-
        is_building(Arg).

can_modify(moderate,damage).

can_modify(oncoming,Arg) :- is_vehicle(Arg).

can_modify(other,Arg) :-
        is_vehicle(Arg).

can_modify(outside,lane).

can_modify(parked,Arg) :- is_vehicle(Arg).

can_modify(passing,lane).
can_modify(posted,zone).

can_modify(rearview,mirror).

can_modify(screeching,tire).

can_modify(sharp,Arg) :- is_turn(Arg).

can_modify(slick,spot).

can_modify(solid,Arg) :-
        is_phys_obj(Arg).

can_modify(sore,Arg) :- is_body_part(Arg).

can_modify(suddenly,Arg) :-
        is_event(Arg).

can_modify(unable,Arg) :-
        is_person(Arg).

can_modify(west,Arg) :-
        is_phys_obj(Arg).

can_modify(yellow,Arg) :-
        is_phys_obj(Arg).

can_modify(Qual,Arg) :-
        is_moving_direction(Qual),
        is_vehicle(Arg).

can_modify(Qual,Arg) :-
        is_moving_direction(Qual),
        is_person(Arg).

can_modify(Qual,Arg) :-
        is_moving_direction(Qual),
        is_street(Arg).

can_modify(Num,Arg) :-
        is_number(Num),
        is_phys_obj(Arg).
can_modify(Num,Arg) :-
```

```
        is_number(Num),
        is_measure(Arg).

%%* Adjectival Prepositions

can_modify(partitive,Arg) :-
        is_object(Arg).

%%* Appositives
%%  can_modify(Arg,Arg).


%%* Adverbs

can_modify(also,Arg) :-
        is_event(Arg).

can_modify(at,Arg) :-
        is_event(Arg).

can_modify(before,Arg) :-
        is_event(Arg).

can_modify(dest,Arg) :-
        is_event(Arg).

can_modify(dir,Arg) :-
        is_event(Arg).

can_modify(for,Arg) :-
        is_event(Arg).

can_modify(mph,drive).

can_modify(not,Arg) :-
        is_event(Arg).

can_modify(onto,Arg) :-
        is_street(Arg).
can_modify(onto,Arg) :-
        is_change_of_loc_action(Arg).

can_modify(over,Arg) :-
        is_event(Arg).
```

```
can_modify(past,Arg) :-
        is_event(Arg).

can_modify(path_change,Arg) :-
        is_event(Arg).

can_modify(severely,Arg) :-
        is_property(Arg).

can_modify(simultaneouswith,Arg) :-
        is_event(Arg).

can_modify(slightly,Arg) :-
        is_property(Arg).

can_modify(then,Arg) :-
        is_event(Arg).

can_modify(very,Arg) :-
        is_property(Arg).

can_modify(wednesday,Arg) :-
        is_event(Arg).

can_modify(when,Arg) :-
        is_event(Arg).

can_modify(while,Arg):-
        is_event(Arg).

can_modify(cover,road).


%% Any action can modify any other action (in a purpose clause)

can_modify(approach,hasproperty).

can_modify(arrive,come).

can_modify(avoid,swerve).

can_modify(back,slow).
can_modify(brace,have).

can_modify(exit,need).

can_modify(turn,pull).

can_modify(go,pull).

can_modify(realize,have).

can_modify(yield,hasproperty).
```

C.2 Semantic Type Information

```
is_name_of(beckman,building).
is_name_of(flb,building).
is_name_of(saintanthonyhospital,building).

is_name_of(howellasphalt,company).

is_name_of(edwardladewick,person).
is_name_of(meuser,person).
is_name_of(officergeorgewissmiller,person).
%is_name_of(num1,person).
%is_name_of(num2,person).
%is_name_of(num3,person).

is_name_of(effingham,town).
is_name_of(urbana,town).
is_name_of(tavernier,town).

is_name_of(busey,street).
is_name_of(clay,street).
is_name_of(claystreet,street).
is_name_of(devonshireroad,street).
is_name_of(duncanroad,street).
is_name_of(fourthstreet,street).
is_name_of(fourth,street).
is_name_of(green,street).
is_name_of(gregory,street).
is_name_of(vinestreet,street).
is_name_of(vine,street).
is_name_of(elm,street).
is_name_of(vermillionst,street).
is_name_of(lincoln,street).
is_name_of(kirby,street).
is_name_of(neil,street).
is_name_of(i74,street).
is_name_of(interstate57,street).

is_name_of(num1,vehicle).
is_name_of(num2,vehicle).
is_name_of(num3,vehicle).

is_action(become).
is_action(brake).
is_action(complete).
is_action(disobey).
is_action(dodge).
is_action(enter).
is_action(go).
is_action(hit).
is_action(leave).
is_action(lose).
is_action(see).
is_action(show).
is_action(stop).
is_action(strike).
is_action(turn).

is_area(zone).
is_area(street).

is_body_part(foot).
is_body_part(knee).
is_body_part(lip).
is_body_part(neck).
is_body_part(nose).

is_building(building).
is_building(dealership).
is_building(hospital).

is_change_of_loc(dir).
is_change_of_loc(into).
is_change_of_loc(outof).
is_change_of_loc(through).
is_change_of_loc(up).

is_change_of_loc_action(come).
is_change_of_loc_action(pull).
is_change_of_loc_action(turn).

is_company(Arg) :-
        is_name_of(Arg,company).

is_commodity(time).

is_direction(north).
is_direction(south).
is_direction(east).
is_direction(west).
is_direction(left).
is_direction(right).
is_direction(forward).

is_event(accident).
is_event(collision).
is_event(failure).
is_event(hasproperty).
is_event(Arg) :-
        is_action(Arg).

is_information(identity).

is_maneuver(stop).

is_measure(block).
is_measure(foot).
is_measure(mile).

is_motion(carry).
is_motion(slide).

is_moving_direction(northbound).
is_moving_direction(southbound).
is_moving_direction(eastbound).
is_moving_direction(westbound).
```

```
is_number(three).
is_number(five).
is_number(twenty).
is_number(twelvehundred).

is_object(Arg) :-
        is_body_part(Arg).
is_object(Arg) :-
        is_person(Arg).
is_object(Arg) :-
        is_phys_object(Arg).
is_object(Arg) :-
        is_street(Arg).

is_person(claimant).
is_person(driver).
is_person(insured).
is_person(owner).
is_person(party).
is_person(people).
is_person(person).
is_person(passenger).
is_person(witness).
is_person(i).
is_person(he).
is_person(she).

is_phys_obj(block).
is_phys_obj(corner).
is_phys_obj(curb).
is_phys_obj(fender).
is_phys_obj(pole).
is_phys_obj(rear).
is_phys_obj(sheet).
is_phys_obj(side).
is_phys_obj(stopsign).
is_phys_obj(tire).
is_phys_obj(Arg) :-
        is_vehicle(Arg).

is_position(end).

is_property(belligerent).
is_property(complete).
is_property(dead).
is_property(disinterested).
is_property(early).
is_property(eastbound).
is_property(grassy).
is_property(green).
is_property(here).
is_property(icy).
is_property(local).
is_property(moderate).
is_property(northbound).
is_property(old).
is_property(red).
is_property(sharp).
is_property(slick).
is_property(solid).
is_property(sore).
is_property(southbound).
is_property(westbound).
is_property(yellow).

is_proposition(hasproperty).
is_proposition(Arg) :-
        is_event(Arg).

is_street(lane).
is_street(road).
is_street(street).
is_street(Arg) :-
        is_name_of(Arg,street).

is_tcd(sign).
is_tcd(signal).

is_time_chunk(afternoon).
is_time_chunk(evening).
is_time_chunk(morning).

is_town(town).
is_town(Arg) :-
        is_name_of(Arg,town).

is_turn(left).
is_turn(right).
is_turn(turn).

is_vehicle(cadillac).
is_vehicle(car).
is_vehicle(grinder).
is_vehicle(traffic).
is_vehicle(truck).
is_vehicle(vehicle).
is_vehicle(Arg) :-
        is_name_of(Arg,vehicle).
```

Appendix D

Learning Procedure Code

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%  learner.pl
%%  This file contains the predicates for Dale Russell's
%%  learning component added to Unicorn.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

D.1 Starting a Learning Session

```
%%  This function is used to set up a learning session.  In a
%%  learning session, we want to remember the successful hypotheses
%%  for a new word, and unify them with previous successful
%%  hypotheses for the same word in the same learning session, to
%%  narrow down the possibilities for the actual lexical entry of
%%  the word.
%%  start_session initializes the files in which the new word is
%%  to be stored.

start_session :-
        unglarf,
        open(newlexentry,write,Stream),
        write(Stream,'□.'),
        nl(Stream),
        close(Stream).
```

D.2 Making a List of Candidates

```
%%  The purpose of make_candidate_list is to create a list of candidate
%%  categories from the list of restricted predictors.  This
%%  essentially means weeding out duplicates.  Exact duplicates are
%%  already caught by add_predictor, but we also want to weed out
%%  cases where candidates are the same except that one has "t"
%%  (anything) where the other has "*?*" (unknown).

%%  Termination condition -- the list of restricted predictors is empty.
%%  The list is ordered such that the solutions that involve fewer PS
%%  rules, i.e. a simpler tree structure, come first.  However, this
%%  makes no difference if the system is looking for all solutions
%%  anyway.

make_candidate_list([],CandidatesSoFar,CandidatesSoFar).

%%  Ignore a candidate if the bar level is known not to be zero.
make_candidate_list([[_,Bar|_]|RestRPreds],
                    CandidatesSoFar,
                    CandidateList) :-
        member(Bar,[one,two]),!,
        make_candidate_list(RestRPreds,CandidatesSoFar,CandidateList).

%%  Ignore a candidate if it (or something equivalent to it) is already
%%  on the list.
make_candidate_list([FirstRPred|RestRPreds],CandidatesSoFar,Candidate_List) :-
        already_candidate(FirstRPred,CandidatesSoFar),
        make_candidate_list(RestRPreds,CandidatesSoFar,Candidate_List).

%%  Default -- add the new candidate to the candidate list.
make_candidate_list([FirstRPred|RestRPreds],CandidatesSoFar,Candidate_List) :-
        make_candidate_list(RestRPreds,
                            [FirstRPred|CandidatesSoFar],
                            Candidate_List).


%%  Returns true iff FirstRPred, the first restricted predictor, is
%%  equivalent to something on the candidates list.

already_candidate(FirstRPred,[FirstCandidate|_]) :-
        same_candidates(FirstRPred,FirstCandidate).

already_candidate(FirstRPred,[_|RestCandidates]) :-
```

```
        already_candidate(FirstRPred,RestCandidates).


%%  Returns true iff two candidates (lists of restricted predictors)
%%  are the same, where *?* counts as the same as anything else.

same_candidates([],[]).

same_candidates([Atom|RestRP],[Atom|RestCand]) :-
        same_candidates(RestRP,RestCand).

same_candidates([*?*|RestRP],[_Atom|RestCand]) :-
        same_candidates(RestRP,RestCand).

same_candidates([_Atom|RestRP],[*?*|RestCand]) :-
        same_candidates(RestRP,RestCand).
```

D.3 Trying the Candidates One at a Time

```
%% The purpose of try_cats is to go through the list of candidates
%% for the unknown word one at a time, and attempt to parse the
%% remainder of the input with each.  The arguments beginning with
%% "Orig" hold the values of the corresponding arguments at the
%% point where the unknown became the first word of the input.

%% The result that this function returns is a disjunctive list of
%% possible meanings for this word after analyzing this piece of input.
%% Each disjunct consists of an upper bound of info, and a set of
%% corresponding lower bounds.  The upper bound is the info known
%% to be in the new lexical item, because it is given by the restricted
%% predictors, from info specified in the restrictor.  The lower bounds
%% are instantiations of this upper bound in actual parses, and represent
%% the maximal amount of info that might be in the lexical item.

%% Termination condition.  If there are no candidates, there can
%% be no successes.
try_cats(_OrigF,_OrigStateSets,_OrigCurrentSets,
         _OrigPredictors,_OrigRestrictedPredictors,_OrigScansList,
         _Input,[],[]).

try_cats(OrigF,OrigStateSets,OrigCurrentSets,
         OrigPredictors,OrigRestrictedPredictors,OrigScansList,
         Input,[FirstCand|RestCands],Successes) :-
```

```
        nl,write('Hypothesizing that new word is of category '),
        write(FirstCand),nl,nl,
        construct_lex_gram(1001,Input,FirstCand,Compiled_newword),
        temp_assert(Compiled_newword,NewRule),
        %% Call add_states, with all parameters reset to what they were
        %% at the point where the unknown word first became the initial
        %% word of the input.
        add_states(OrigF,OrigStateSets,OrigCurrentSets,
                   OrigPredictors,OrigRestrictedPredictors,OrigScansList,
           OrigF,OrigStateSets,OrigCurrentSets,
           OrigPredictors,OrigRestrictedPredictors,OrigScansList,
           TempTerminationSet,Input),
        terminate(TempTerminationSet),
        temp_unassert(Compiled_newword),
        try_cats(OrigF,OrigStateSets,OrigCurrentSets,
                 OrigPredictors,OrigRestrictedPredictors,OrigScansList,
                 Input,RestCands,NewSuccesses),
        if(TempTerminationSet = [],
           Successes = NewSuccesses,
           (instantiate_rules(NewRule,TempTerminationSet,[],InstanRules),
            make_ub(NewRule,NewUpperBound),
            Successes = [[NewUpperBound,InstanRules]|NewSuccesses])).


%% Opens up the one-word grammar of a new lexical item,
%% converts it to a rule, and asserts that rule.
temp_assert(File,Rule) :-
        open(File,read,Stream),
        read(Stream,Rule),
        close(Stream),
        nl,write(Rule),nl,
        assert(Rule).


%% Undoes the work of temp_assert, retracting the
%% assertion made by a one-word grammar.
temp_unassert(File) :-
        open(File,read,Stream),
        read(Stream,Rule),
        close(Stream),
        retract(Rule).
```

D.3.1 Constructing a One-Word Grammar

```
%% Make a one-word grammar for the new word.  Some of this is the same for
%% all words, some is determined by the syntactic category, and some is
%% determined by the information passed in, which was specified by the
%% restrictor.
%% The function construct_lex_gram, along with some
%% auxiliary predicates, reads information from the restrictor, and so
%% constructs the lexical grammar automatically.  For each path on the
%% restrictor, if its value is not *?*, convert it to a line of the
%% lexical grammar.  This enables the learning system to work with
%% any grammar, so long as a reasonable restrictor is given.

%% This function specifies the paths that are the same for words of all
%% syntactic categories.
construct_lex_gram(RuleNum,[Unknown|_],PathValues,Compiled_newword) :-
        paths(Paths),
        nl,write('Acq Restrictor: '),write(Paths),nl,
        PathValues = [Cat|_],
        open(newword,write,Stream),
        write(Stream, 'Word '),
        write(Stream, Unknown),
        write(Stream, ':'),
        nl(Stream),
        write(Stream, '<pecParseTree daughters word> = '),
        write(Stream, Unknown),
        nl(Stream),
        write(Stream, '<pecParseTree rule_number> = rn_'),
        write(Stream, RuleNum),
        nl(Stream),
        write(Stream, '<phon list first> = '),
        write(Stream, Unknown),
        nl(Stream),
        write(Stream, '<phon list rest> = <phon eol>'),
        add_content_paths(Cat,Stream),
        add_restrictor_paths(Paths,PathValues,Stream),
        close(Stream),
        compile_grammar_name(newword,Compiled_newword).


%% The function add_content_paths specifies the paths making up the semantic
%% content of the word.  These are different for each syntactic category,
%% but known to be constant within each category, except for verbs.
```

BEST COPY AVAILABLE

```
add_content_paths(adj,Stream) :-
        nl(Stream),
        write(Stream, '<synsem local content index> = '),
        write(Stream, '<synsem local category head mod
                        synsem local content index>'),
        nl(Stream),
        write(Stream, '<synsem local content restr first> = '),
        write(Stream, '<synsem local category head mod
                        synsem local content restr first>'),
        nl(Stream),
        write(Stream, '<synsem local content restr rest first arg> = '),
        write(Stream, '<synsem local category head mod
                        synsem local content index>'),
        nl(Stream),
        write(Stream, '<synsem local content restr rest first relation> =
                       Glarf'),
        nl(Stream),
        write(Stream, '<synsem local content restr rest rest> = '),
        write(Stream, '<synsem local category head mod
                        synsem local content restr rest>').

add_content_paths(adv,Stream) :-
        nl(Stream),
        write(Stream, '<synsem local content psoa> = '),
        write(Stream, '<synsem local category head mod
                        synsem local content psoa>'),
        nl(Stream),
        write(Stream, '<synsem local content restr first soaarg> = '),
        write(Stream, '<synsem local category head mod
                        synsem local content psoa>'),
        nl(Stream),
        write(Stream, '<synsem local content restr rest> = '),
        write(Stream, '<synsem local category head mod
                        synsem local content restr>'),
        nl(Stream),
        write(Stream, '<synsem local content restr first relation> = Glarf').

add_content_paths(det,Stream) :-
        nl(Stream),
        write(Stream, '<synsem local content det> = Glarf'),
        nl(Stream),
        write(Stream, '<synsem local content restind> = '),
        write(Stream, '<synsem local category head spec
                        synsem local content>').
```

```
add_content_paths(n,Stream) :-
        nl(Stream),
        write(Stream, '<synsem local content restr first instance> = '),
        write(Stream, '<synsem local content index>'),
        nl(Stream),
        write(Stream, '<synsem local content restr first relation> = Glarf').

add_content_paths(v,Stream) :-
        nl(Stream),
        write(Stream, '<synsem local content psoa relation> = Glarf').

add_content_paths(_,_).


%%  add_restrictor takes as input a list of paths and the corresponding
%%  list of values along those paths found by the restricted predictor.
%%  For each pair of a path and its value, it writes (onto the given stream)
%%  a path equation for the lexical grammar.

%%  When there are no more paths, add a period to the end.
add_restrictor_paths([],_,Stream) :-
        write(Stream, '.'),
        nl(Stream).

%%  Special case.  If the bar level is unknown, stipulate it to be zero.
add_restrictor_paths([[synsem,local,category,bar]|RestPaths],
                     ['*?*'|RestPathValues],
                     Stream) :-
        nl(Stream),
        write(Stream, '<synsem local category bar> = zero'),
        add_restrictor_paths(RestPaths,RestPathValues,Stream).

%%  Don't add a path if its value is unknown.
add_restrictor_paths([_FirstPath|RestPaths],['*?*'|RestPathValues],Stream) :-
        add_restrictor_paths(RestPaths,RestPathValues,Stream).

%%  Normal case.  Add a path to the one-word grammar.
add_restrictor_paths([FirstPath|RestPaths],
                     [FirstPathValue|RestPathValues],
                     Stream) :-
        nl(Stream),
        write(Stream, '<'),
        write_rpath(Stream,FirstPath),
        write(Stream, '> = '),
        write(Stream,FirstPathValue),
        add_restrictor_paths(RestPaths,RestPathValues,Stream).


%% Given a single path, expressed as a list, write it onto the stream
%% as a path equation.

write_rpath(_Stream,[]).

write_rpath(Stream,[LastNode]) :-
        write(Stream,LastNode).

write_rpath(Stream,[FirstNode|RestNodes]) :-
        write(Stream,FirstNode),
        write(Stream,' '),
        write_rpath(Stream,RestNodes).
```

D.3.2 Instantiating Successful Lexical Hypotheses

```
%% An upper bound is just like an uninstantiated rule, except that
%% wherever the dag of the rule has a variable with no value in the
%% re-entrancy list, the upper bound has to have a re-entrancy
%% binding that value to t, for "anything."

make_ub(lex_rule([Length,Phon,RuleDag]),
        lex_rule([Length,Phon,UpperDag])) :-
        bind_off(RuleDag,UpperDag).


%% Termination condition for binding off an upper bound.
bind_off([[],RL],[[],RL]).

%% The value of the first path is not bound to anything in the RL.
%% In the upper bound, the path and value are exactly the same, but
%% the RL is augmented with a reentrancy binding this variable to t.
bind_off([[[FirstAtt,FirstVal]|RestPaths],RuleRL],
         [[[FirstAtt,FirstVal]|RestPaths],UpperRL]) :-
        var(FirstVal),
        get_val(FirstVal,RuleRL,RLVal),
        if(var(RLVal),
          (append(RuleRL,[[FirstVal,t]],NewRL),
           bind_off([RestPaths,NewRL],[RestPaths,UpperRL])),
          (bind_off([[[FirstAtt,RLVal]|RestPaths],RuleRL],
                    [[[FirstAtt,RLVal]|RestPaths],UpperRL]))).

%%  If the value of the first path is bound to an atom, continue
%%  with the rest of the dag.
bind_off([[[FirstAtt,FirstVal]|RestPaths],RuleRL],
         [[[FirstAtt,FirstVal]|RestPaths],UpperRL]) :-
        atom(FirstVal),
        bind_off([RestPaths,RuleRL],[RestPaths,UpperRL]).

%%  The value of the first path is a list, do a recursive call.
bind_off([[[FirstAtt,FirstVal]|RestPaths],RuleRL],
         [[[FirstAtt,FirstVal]|RestPaths],UpperRL]) :-
        bind_off([FirstVal,RuleRL],[FirstVal,NewRL]),
        bind_off([RestPaths,NewRL],[RestPaths,UpperRL]).


%%  This function oversees the collection of instantiated lexical
%%  rules.  As each successful instantiation is collected by
%%  instantiate_rule, it is compared to those collected so far,
%%  to weed out duplicates.

instantiate_rules(_Lex_rule,[],IRulesSofar,IRulesSofar).

instantiate_rules(Lex_rule,
                  [FirstTermination|RestTerminations],
                  IRulesSofar,
                  NewIRules) :-
        instantiate_rule(Lex_rule,FirstTermination,NewIRule),
        %% Check whether the new instantiated rule, NewIRule, is
        %% equivalent to any of the IRulesSofar.  If so, do nothing.
        %% Otherwise, add the NewIRule to the end of the list of
        %% IRulesSofar.
        if(already_on_lex_list(NewIRule,IRulesSofar),
           (instantiate_rules(Lex_rule,
                              RestTerminations,
                              IRulesSofar,
                              NewIRules)),
           (instantiate_rules(Lex_rule,
                              RestTerminations,
                              [NewIRule|IRulesSofar],
                              NewIRules))).


%%  This function creates an instantiated rule by retrieving that
%% portion of the successful termination that uses the new hypothesis.
%% This portion is identified by the Rule Number given as input.  The
%% function calls match_term_fs to look through a successful parse for
%% an instance of the new lexical item.  When it is found, it create
%% an instantiated lexical rule, whose Feature Structure is the FS of
%% that word in the successful parse, and whose Reentrancy List is
%% that portion of the RL of the Termination set whose variables
%% appear in the lexical FS.
%% Right now, I'm using the RL of the whole parse as the RL of the
%% instantiated lexical rule.  This will include some unnecessary
%% reentrancies, but it's very hard to pick out just those that
%% are needed.  This could be a future task.

instantiate_rule(lex_rule([Length,
                           Phon,
                           [RuleFS,_RuleRL]]),
                 [TermFS,TermRL],
                 lex_rule([Length,
                           Phon,
                           [InstanFS,TermRL]])) :-
        get_rn(RuleFS,RuleNum),
        nl,write(RuleNum),nl,
        match_term_fs(RuleNum,[TermFS,TermRL],InstanFS).

%% Look through the dag of a successful parse for the new lexical
%% rule and retrieve its rule number.

get_rn([],'Error').
get_rn([[pecParseTree,[_Dtrs,[rule_number,RuleNum]]]|_],RuleNum).
get_rn([_First|Rest],RuleNum) :-
        get_rn(Rest,RuleNum).


%% Look through the dag of the successful parse, and find the
%% portion of the feature structure licensed by the rule number
%% of the newly learned word.  This will give us the instantiated
%% feature structure of that word as it was actually used in
%% generating a successful parse.

%% Terminate on success.  Found the desired rule number in a
%% lexical rule.
%% Note:  it would be better to generalize this so that
%% pecParseTree doesn't have to be the second path in the dag.
match_term_fs(RuleNum,
              [TermFS,_TermRL],
              _RLSeen,
              TermFS) :-
        TermFS = [[maj,_Syncat],
                  [pecParseTree,[_Dtrs,[rule_number,RuleNum]]]|_].

%%  Looking at a node with daughters.  Obviously, the new word
%%  isn't here, so we look at its daughters.
match_term_fs(RuleNum,
              [[_Dtrs,_Maj,[pecParseTree,[[daughters,PecDtrs]|_]]|_],
               TermRL],
               RLSeen,
              InstanFS) :-
        nl,write('Seen; '),write(RLSeen),nl,
        match_term_fs(RuleNum,[PecDtrs,TermRL],RLSeen,InstanFS).

%% The node we're looking at has a reentrancy number at the
%% end of its path.  If this number is one that has already been
%% followed, don't follow it again, or you'll get into an infinite
%% loop.  Otherwise, get the value from the reentrancy list
%% and keep looking, with the value in place of the number.

match_term_fs(RuleNum,[[[_Cat,Var]],TermRL],RLSeen,InstanFS) :-
        var(Var),
        nl,write('Seen; '),write(RLSeen),nl,
        if(member_var(Var,RLSeen),
          fail,
          (get_val(Var,TermRL,Val),
           match_term_fs(RuleNum,[Val,TermRL],[Var|RLSeen],InstanFS))).

%% There are two daughters, each with a reentrancy at the end
%% of its path.  Try the left first, and if that doesn't work,
%% try the right.
%% Note: May need another version of this for three daughters.
match_term_fs(RuleNum,
              [[[_Cat1,Var1],[_Cat2,Var2]],TermRL],
              RLSeen,
              InstanFS) :-
        var(Var1),var(Var2),
        nl,write('Seen; '),write(RLSeen),nl,
        if(member_var(Var1,RLSeen),
          fail,
          (get_val(Var1,TermRL,Val1),
           match_term_fs(RuleNum,[Val1,TermRL],[Var1|RLSeen],InstanFS))).
match_term_fs(RuleNum,
              [[[_Cat1,Var1],[_Cat2,Var2]],TermRL],
              RLSeen,
              InstanFS) :-
        var(Var1),var(Var2),
        nl,write('Seen; '),write(RLSeen),nl,
        if(member_var(Var2,RLSeen),
          fail,
          (get_val(Var2,TermRL,Val2),
           match_term_fs(RuleNum,[Val2,TermRL],[Var2|RLSeen],InstanFS))).

member_var(Var1,[Var2|_RestVars]) :-
        Var1 == Var2.

member_var(Var1,[_First|RestVars]) :-
        member_var(Var1,RestVars).
```

D.4 Reading and Writing Hypotheses to Files

```
%%  Get the hypothesized definitions for the unknown word from a
%%  file.
get_hypotheses(_Unknown,Hypotheses) :-
        open(newlexentry,read,Stream),
        read(Stream,Hypotheses),
        close(Stream).


%%  For a given unknown word, pull out the definitions for that
%%  word hypothesized previously in the learning session.  Return
%%  a list of those definitions, and the remainder of the hypotheses;
%%  i.e., those about other unknown words.

%%  Termination Condition
get_previous_defs(_Unknown,[],[],[]).

%%  First hyp is a def of the unknown.
get_previous_defs(Unknown,
                  [FirstHyp|RestHyps],
                  [FirstHyp|RestDefs],
                  Remainder) :-
        FirstHyp = [Unknown|_],
        get_previous_defs(Unknown,RestHyps,RestDefs,Remainder).
```

```
%% First hyp is not a def of the unknown.
get_previous_defs(Unknown,
                  [FirstHyp|RestHyps],
                  WordDefs,
                  [FirstHyp|RestRemainder]) :-
        get_previous_defs(Unknown,RestHyps,WordDefs,RestRemainder).


%% Replace the current list of hypothesized lexical items with
%% the updated list.
put_hypotheses(Definition) :-
        open(newlexentry,write,Stream),
        write(Stream,Definition),
        write(Stream,'.'),
        nl(Stream),
        close(Stream).
```

D.5 Combining New and Previous Hypothesized Definitions

```
%% combine(Successes,PreviousDefinitions,UpdatedDefinitions).
%% The inputs to combine are Successes and Previous Definitions.
%% The output is an updated list of definitions.

%% Successes is an unknown and the list of hypotheses for the
%% unknown constructed from the current piece of input.
%% A hypothesis is an upper bound and a set of lower bounds.

%% PrevDefs is a list, where each element consists of an unknown and
%% a list of hypotheses from earlier input in the learning session.
%% The reason PrevDefs has to be a list of lists is that we may
%% have encountered two tokens of "glarf" that cannot be the same
%% lexical item; one is an adjective, the other is either a verb
%% or a noun.  If the new token is either a verb or a determiner,
%% it will combine with the second existing entry, but not the
%% first.

%% The way this function is written, it will sometimes do the
%% wrong thing, but only on deliberately perverse data, where
%% I suspect that people would also do the wrong thing, or not
%% be able to say what the right thing should be.
%% For example, suppose that you're learning the surface form
%% "train."  You get two successive input sentence, the first
%%  with "train" used as a verb, similar in meaning to "educate,"
%%  the second with "train" as a noun, in the sense of "locomotive."
%%  Suppose further that on the first instance, we can determine
%%  that the word is either a verb or an adverb, in the second,
%%  either a noun or an adjective. Now we get a third input
%%  sentence with "train," meaning unknown, where it can be either
%%  a verb or a noun.  What do we do?  I don't know the right thing.
%%  What this function will do is to unify it with the first entry
%%  it sees, and ignore all others.

%%  If there are no previous definitions, just store the new one.
combine(Successes,[],[Successes]).

%%  ThereOverlap between Successes and FirstPrevDef:
combine([Unknown,NewHyps],
        [[Unknown,PrevHyps]|RestPrevDefs],
        [[Unknown,MergedDef]|RestPrevDefs]) :-
        merge([Unknown,NewHyps],[Unknown,PrevHyps],MergedDef),
        nl,write('MergedDef: '),write(MergedDef),nl,
        MergedDef \== [].

%%  Otherwise -- there is no overlap between Successes and FirstPrevDef:
%%  Assume that Successes represents a sense of the word unrelated
%%  to FirstPrevDef.  Try to combine Successes with other senses
%%  of the word.
combine(Successes,[FirstPrevDef|RestPrevDefs],[FirstPrevDef|RestNewDefs]) :-
        combine(Successes,RestPrevDefs,RestNewDefs).


%% merge([Unknown,NewHyps],[Unknown,PrevHyps],CommonUBs,MergedDef).
%%  The MergedDef consists of only those hypotheses with upper
%%  bounds produced by successfully unifying those of the new
%%  hypotheses and the previous hypotheses.  For each of
%%  these hypotheses, the upper bound of the MergedDef should
%%  be the unification of those of the new and previous hypotheses,
%%  while the lower bounds are the intersection of those of the two.

%%  Termination condition -- no more new hypotheses.
merge([Unknown,[]],[Unknown,_PrevHyps],[]).

%%  PrevLower is the lower bound corresponding to the
%%  upper bound of PrevHyps which successfully unifies
%%  with the UB of the FirstNewUpper.
%%  Take the cross-product of the two sets of lower bounds.
```

```
merge([Unknown,[[FirstNewUpper,FirstNewLower]|RestNewHyps]],
       [Unknown,PrevHyps],
       [[UnifUB,MergedLower]|RestMDef]) :-
         unif_member_ub(FirstNewUpper,PrevHyps,PrevLower,UnifUB),
         UnifUB \== [],!,
         nl,write('Unified Upper Bound: '),write(UnifUB),nl,
         cross_lb_sets(FirstNewLower,PrevLower,UnifUB,MergedLower),
         merge([Unknown,RestNewHyps],[Unknown,PrevHyps],RestMDef).

%% Otherwise -- UB of the first hyp does NOT unify with any UB in
%% PrevHyps.  Discard this upper bound, and try others.

merge([Unknown,[_FirstHyp|RestHyps]],[Unknown,PrevHyps],MergedDef) :-
         merge([Unknown,RestHyps],[Unknown,PrevHyps],MergedDef).


%%  Returns the first successful unification of an upper bound of
%%  the new hypothesized definition with one of the upper bounds
%%  of previously hypothesized definitions of the same word.

unif_member_ub(_FirstCurUpper,[],[],[]).

%% Unification succeeds
unif_member_ub(lex_rule([Length,Phon,CurDag]),
                [[lex_rule([Length,Phon,PrevDag]),FirstPrevLower]|
                 _RestPrevHyps],
                FirstPrevLower,
                lex_rule([Length,Phon,UnifDag])) :-
        d_union(CurDag,PrevDag,UnifDag).


%% Otherwise (unification didn't succeed).
unif_member_ub(FirstCurUpper,
                [_FirstPrevHyp|RestPrevHyps],
                PrevLower,
                UnifFirstUpper) :-
        unif_member_ub(FirstCurUpper,RestPrevHyps,PrevLower,UnifFirstUpper).


%%  Take the cross-comparison of two sets of LBs.  For each pairwise
%%  combination, we want to find the information that they have in
%%  common.  If this information is greater than or equal to that in
%%  the unified upper bound, keep it.  Otherwise, disgard it.
%%  cross_lb_sets(NewLower,PrevLower,UnifUB,MergedLower).

cross_lb_sets(_NewLower,[],_,[]).

cross_lb_sets(NewLower,[FirstPrevLower|RestPrevLower],UnifUB,MergedLower) :-
        cross_lb_set(NewLower,FirstPrevLower,UnifUB,FirstMergedLower),
        cross_lb_sets(NewLower,RestPrevLower,UnifUB,RestMergedLower),
        append_lex_without_duplicates(FirstMergedLower,
                                      RestMergedLower,
                                      MergedLower).


%%  Cross-compare a set of LBs and a single LB.
%%  cross_lb_set(NewLower,FirstPrevLower,UnifUB,MergedLBSet).

cross_lb_set([],_FirstPrevLower,_,[]).

cross_lb_set([FirstNewLower|RestNewLower],FirstPrevLower,UnifUB,MergedLBSet) :-
        cross_lb_set(RestNewLower,FirstPrevLower,UnifUB,RestMergedLBSet),
        cross_lb(FirstNewLower,FirstPrevLower,FirstMergedLBSet),!,
        %% Don't add a product if it already exists as one of the
        %% previously found lower bounds.
        nl,write('Start checking'),nl,
        if(already_on_lex_list(FirstMergedLBSet,RestMergedLBSet),
           MergedLBSet = RestMergedLBSet,
           %% Don't want to add products that contradict the upper bound;
           %% i.e., don't add a product if the upper bound has any path
           %% that the product doesn't.
           (UnifUB = lex_rule([_ULength,_UPhon,UDag]),
            FirstMergedLBSet = lex_rule([_MLength,_MPhon,MDag]),
            if(has_additional_path(UDag,MDag),
               MergedLBSet = RestMergedLBSet,
               MergedLBSet = [FirstMergedLBSet|RestMergedLBSet]))).


%%  Merge a single LB with another single LB.

cross_lb(lex_rule([Length,Phon,[NewFS,NewRL]]),
         lex_rule([Length,Phon,[PrevFS,PrevRL]]),
         lex_rule([Length,Phon,[MergedFS,MergedRL]])) :-
        %% Get a FS that has all the info of both NewFS and PrevFS,
        %% except where they contradict, and then have nothing.
        cross_fs(NewFS,NewRL,PrevFS,PrevRL,[],MergedFS,MergedRL).
```

```
%% In case of failure, return the empty set.
cross_lb(_,_,[]).


%%  Make a new dag which represents the information shared by
%%  the two input dags, New and Prev.
%%  cross_fs(NewFS,NewRL,PrevFS,PrevRL,MergedFS,MergedRL).

%% No paths left in one of the feature structures, so we're done.
cross_fs([],_NewRL,_PrevFS,_PrevRL,_RLSeen,[],[]).
cross_fs(_NewFS,_NewRL,[],_PrevRL,_RLSeen,[],[]).

%%  Three possibilities: NewAtt and PrevAtt are the same.
%%                       NewAtt comes alphabetically first.
%%                       PrevAtt comes alphabetically first.
cross_fs([[Att,NewVal]|RestNewPaths],NewRL,
         [[Att,PrevVal]|RestPrevPaths],PrevRL,
         RLSeen,
         MergedFS,MergedRL) :-
        get_merged_val(NewVal,NewRL,PrevVal,PrevRL,RLSeen,
                       MergedVal,NewMergedRL),!,
        nl,write('Seen1; '),write(RLSeen),nl,
        if(MergedVal == [],
           cross_fs(RestNewPaths,NewRL,RestPrevPaths,PrevRL,
                    RLSeen,MergedFS,MergedRL),
           (cross_fs(RestNewPaths,NewRL,RestPrevPaths,PrevRL,
                     RLSeen,RestMergedFS,RestMergedRL),
            MergedFS = [[Att,MergedVal]|RestMergedFS],
            append_rl_without_duplicates(NewMergedRL,RestMergedRL,MergedRL))).

cross_fs([[NewAtt,_NewVal]|RestNewPaths],NewRL,
         [[PrevAtt,PrevVal]|RestPrevPaths],PrevRL,
         RLSeen,
         MergedFS,MergedRL) :-
        NewAtt @< PrevAtt,!,
        cross_fs(RestNewPaths,NewRL,
                 [[PrevAtt,PrevVal]|RestPrevPaths],PrevRL,
                 RLSeen,
                 MergedFS,MergedRL).

cross_fs([FirstNewPath|RestNewPaths],NewRL,
         [_FirstPrevPath|RestPrevPaths],PrevRL,
         RLSeen,
         MergedFS,MergedRL) :-
        cross_fs([FirstNewPath|RestNewPaths],NewRL,
                 RestPrevPaths,PrevRL,
                 RLSeen,
                 MergedFS,MergedRL).

get_intersection([],_,[]).
get_intersection(_,[],[]).
get_intersection([FirstSetOne|RestSetOne],SetTwo,[FirstSetOne|RestIntersection]) :-
        member(FirstSetOne,SetTwo),
        get_intersection(RestSetOne,SetTwo,RestIntersection).
get_intersection([_FirstSetOne|RestSetOne],SetTwo,RestIntersection) :-
        get_intersection(RestSetOne,SetTwo,RestIntersection).

%% If either value is a variable, it must be preserved in the output,
%% as other variables in either of the structures may be reentrant with
%% one of these input variables.  Therefore, we must add a reentrancy of
%% one of these variables to the other, and put it in the output.

get_merged_val(X,_NewRL,_Y,_PrevRL,RLSeen,X,[]) :-
        var(X),
        nl,write('Seen2; '),write(RLSeen),nl,
        member_var(X,RLSeen),!.

get_merged_val(X,NewRL,Y,PrevRL,RLSeen,X,MergedRL) :-
        var(X),
        nl,write('Seen3; '),write(RLSeen),nl,
        get_val(X,NewRL,NewVal),!,
        get_merged_val(NewVal,NewRL,Y,PrevRL,[X|RLSeen],
                       MergedVal,RestMergedRL),
        MergedVal \== [],!,
        if(X == MergedVal,
          MergedRL = RestMergedRL,
          append_rl_without_duplicates([[X,MergedVal]],RestMergedRL,MergedRL)).

get_merged_val(_NewVal,_NewRL,Y,_PrevRL,RLSeen,Y,[]) :-
        var(Y),
        nl,write('Seen4; '),write(RLSeen),nl,
        member_var(Y,RLSeen),!.

get_merged_val(NewVal,NewRL,Y,PrevRL,RLSeen,Y,MergedRL) :-
        var(Y),
        nl,write('Seen5; '),write(RLSeen),nl,
        nl,write('Y; '),write(Y),nl,
        nl,write('PrevRL; '),write(PrevRL),nl,
        get_val(Y,PrevRL,PrevVal),!,
        nl,write('Seen7; '),write(PrevVal),nl,
        get_merged_val(NewVal,NewRL,PrevVal,PrevRL,
                        [Y|RLSeen],MergedVal,RestMergedRL),
        nl,write('Seen8; '),write(RestMergedRL),nl,
        MergedVal \== [],!,
        nl,write('Seen9; '),write(MergedVal),nl,
        if(Y == MergedVal,
          MergedRL = RestMergedRL,
          append_rl_without_duplicates([[Y,MergedVal]],RestMergedRL,MergedRL)).

%%  If either value is a disjunction, return the intersection of that
%%  disjunction with the set containing the other value.

get_merged_val(disj(NewSet),_NewRL,disj(PrevSet),_PrevRL,
                _RLSeen,disj(IntersectSet),[]) :-
        nl,write('Disjunction '),nl,
        get_intersection(NewSet,PrevSet,IntersectSet),!.

%%  If either value is null, return the empty set, for failure.
get_merged_val([],_,_,_,_,[],[]) :- !.
get_merged_val(_,_,[],_,_,[],[]) :- !.

%%  If one value equals t, return t, the less informative value.
get_merged_val(t,_NewRL,_Y,_PrevRL,_,t,[]) :- !.
get_merged_val(_X,_NewRL,t,_PrevRL,_,t,[]) :- !.

%%  If the two atomic values are the same, return that value.
get_merged_val(X,_,X,_,_,X,[]) :-
        atom(X),!.

%%  If one value is atomic and the other a list, return the
%%  empty set, for failure.
get_merged_val(X,_,Y,_,_,[],[]) :-
        atom(X),
        list(Y),!.
get_merged_val(X,_,Y,_,_,[],[]) :-
        list(X),
        atom(Y),!.

%%  If both values are lists, return their cross-unification.
get_merged_val(NewList,NewRL,PrevList,PrevRL,RLSeen,MergedVal,MergedRL) :-
        list(NewList),
        list(PrevList),
        nl,write('Seen6; '),write(RLSeen),nl,
        cross_fs(NewList,NewRL,PrevList,PrevRL,RLSeen,MergedVal,MergedRL),!.

%%  Otherwise, return the empty set, for failure.
get_merged_val(_X,_,_Y,_,_,[],[]) :- !.


%%  Succeeds iff the first dag contains some path not in the second dag.
%%  This relies on the assumption that all feature structures will have their
%%  paths in alphabetical order.
%% has_additional_path([UnifiedFS,UnifiedRL],
%%                     [MergedFS,MergedRL]).

has_additional_path([[],_URL],_MDag) :- !,fail.

has_additional_path([[[UAtt,_UVal]|_],_URL],[[[MAtt,_MVal],_],_MRL]) :-
        UAtt @< MAtt.

has_additional_path([[[UAtt,UVal]|RestUPaths],URL],
                    [[[MAtt,_MVal]|RestMPaths],MRL]) :-
        UAtt @> MAtt,
        has_additional_path([[[UAtt,UVal]|RestUPaths],URL],[RestMPaths,MRL]).

has_additional_path([[[Att,UVal]|_],URL],[[[Att,MVal]|_],MRL]) :-
        list(UVal),
        list(MVal),
        has_additional_path([UVal,URL],[MVal,MRL]).

has_additional_path([[_FirstUPath|RestUPaths],URL],
                    [[_FirstMPath|RestMPaths],MRL]) :-
        has_additional_path([RestUPaths,URL],[RestMPaths,MRL]).
```

D.6 Utilities

```
%%  Just like append, except that (i) elements that occur in both input
%%  sets occur just once in the output appended set (ii) it is designed
%%  to work specifically with lists of reentrancies.

append_rl_without_duplicates([],Set2,Set2).
append_rl_without_duplicates(Set1,[],Set1).

append_rl_without_duplicates([FirstSet1|RestSet1],
                             Set2,
                          AppendedSet) :-
        already_on_rl(FirstSet1,Set2),
        append_rl_without_duplicates(RestSet1,Set2,AppendedSet).

append_rl_without_duplicates([FirstSet1|RestSet1],
                          Set2,
                          [FirstSet1|RestAppendedSet]) :-
        append_rl_without_duplicates(RestSet1,Set2,RestAppendedSet).


%%  Given a reentrancy and a list of reentrancies, this predicate
%%  succeeds iff the reentrancy is the same as one of the members
%%  of the list.
already_on_rl(_Reentrancy,[]) :- fail,!.
already_on_rl([RNum1,Val1],
              [[RNum2,Val2]|_RestRL]) :-
        RNum1 == RNum2,
        Val1 == Val2.
already_on_rl(Reentrancy,[_FirstRL|RestRL]) :-
        already_on_rl(Reentrancy,RestRL).
already_on_rl([],_).


%%  This function is just like append_without_duplicates above, except
%%  that it is written specifically for appending sets of lex-rules,
%%  because there's a special function, already_on_lex_list, for
%%  determining whether a lex_rule is already on a given list of
%%  lex_rules.

append_lex_without_duplicates([],Set2,Set2).

append_lex_without_duplicates([FirstSet1|RestSet1],
                          Set2,
                          AppendedSet) :-
        already_on_lex_list(FirstSet1,Set2),
        append_lex_without_duplicates(RestSet1,Set2,AppendedSet).

append_lex_without_duplicates([FirstSet1|RestSet1],
                          Set2,
                          [FirstSet1|RestAppendedSet]) :-
        append_lex_without_duplicates(RestSet1,Set2,RestAppendedSet).


%%  Given a lexical rule and a list of lexical rules, this predicate
%% succeeds iff the lexical rule is the same as one of the members
%% of the list.
already_on_lex_list(_NewIRule,[]) :- fail,!.
already_on_lex_list(lex_rule([Length,Phon,NewDag]),
                    [lex_rule([Length,Phon,PrevDag])|_RestIRuleList]) :-
        same_dag(NewDag,PrevDag).
already_on_lex_list(NewIRule,[_FirstIRuleList|RestIRuleList]) :-
        already_on_lex_list(NewIRule,RestIRuleList).
already_on_lex_list([],_).


%% Predicate that decides whether two dags are the same.
%% To be the same, two dags must have the same attributes
%% (in the same order, but this isn't a problem, as our paths
%% are all in alphabetical order), with the same values, when
%% those values are instantiated.

%% If either value is a variable, instantiate it.
%% The variable check is to prevent infinite loops, if a variable
%% has no instantiated value.
same_dag([[[Att,X]|RestNewFS],NewRL],PrevDag) :-
        var(X),
        get_val(X,NewRL,NewVal),
        nonvar(NewVal),!,
        same_dag([[[Att,NewVal]|RestNewFS],NewRL],PrevDag).

same_dag(NewDag,[[[Att,Y]|RestPrevFS],PrevRL]) :-
        var(Y),
        get_val(Y,PrevRL,PrevVal),
        nonvar(PrevVal),!,
        same_dag(NewDag,[[[Att,PrevVal]|RestPrevFS],PrevRL]).

%% If both values are variables with no instantiated values, they
%% count as the same.
same_dag([[[Att,NewVal]|RestNewFS],NewRL],
         [[[Att,PrevVal]|RestPrevFS],PrevRL]) :-
        var(NewVal),
        var(PrevVal),
        same_dag([RestNewFS,NewRL],[RestPrevFS,PrevRL]).

%% Special case of disjunction; the lists of disjuncts must be the same.
same_dag([[[Att,disj(NewDisj)]|RestNewFS],NewRL],
         [[[Att,disj(PrevDisj)]|RestPrevFS],PrevRL]) :-
        same_list(NewDisj,NewRL,PrevDisj,PrevRL),
        same_dag([RestNewFS,NewRL],[RestPrevFS,PrevRL]).

%% Atomic values must be the same.
same_dag([[[Att,NewVal]|RestNewFS],NewRL],
         [[[Att,PrevVal]|RestPrevFS],PrevRL]) :-
        atom(NewVal),
        atom(PrevVal),!,
        NewVal = PrevVal,
        same_dag([RestNewFS,NewRL],[RestPrevFS,PrevRL]).

%% Recursive call for embedded structure.
same_dag([[[Att,NewVal]|RestNewFS],NewRL],
         [[[Att,PrevVal]|RestPrevFS],PrevRL]) :-
        list(NewVal),
        list(PrevVal),!,
        same_dag([NewVal,NewRL],[PrevVal,PrevRL]),
        same_dag([RestNewFS,NewRL],[RestPrevFS,PrevRL]).

same_dag([[],_NewRL],[[],_PrevRL]).


%% Predicate to determine whether two lists are the same.  This is used
%% in the case of disjunction in same_dag.  Disjunctive values may be
%% variables, in which case this predicate instantiates them, then
%% compares the instantiated values.
same_list([],_NewRL,[],_PrevRL).
same_list([FirstNew|RestNew],NewRL,[FirstPrev|RestPrev],PrevRL) :-
        var(FirstNew),
        get_val(FirstNew,NewRL,FirstNewVal),
        same_list([FirstNewVal|RestNew],NewRL,[FirstPrev|RestPrev],PrevRL).
same_list([FirstNew|RestNew],NewRL,[FirstPrev|RestPrev],PrevRL) :-
        var(FirstPrev),
        get_val(FirstPrev,PrevRL,FirstPrevVal),
        same_list([FirstNew|RestNew],NewRL,[FirstPrevVal|RestPrev],PrevRL).
%% The firsts of the two lists are the same.
same_list([First|RestNew],NewRL,[First|RestPrev],PrevRL) :-
        same_list(RestNew,NewRL,RestPrev,PrevRL).
```

Appendix E

Sample Learning Episodes

This appendix contains the output of the learning system for a number of selected inputs, to demonstrate the range of material that the system is able to learn. The first section has one output for each syntactic category in the test grammar. The second section shows how the information from multiple inputs is combined to converge on the definition of a new word.

E.1 Sample Output for Each Syntactic Category

This section contains the output of a learning episode for each syntactic category. This inputs used here were carefully chosen so that with a single input, an unambiguous part of speech would be inferred.

The output of each learning episode is a list of resulting lexical version hyperspaces, one for each new word being learned. With a single unknown in the output, this is always a list of length one. Each lexical version hyperspace consists of a list of lexical version spaces, where each lexical version spaces in turn contains a single upper bound followed by a list of one or more lower bounds.

Each upper or lower bound is in the form of a lexical rule, and is signaled by the string `lex_rule`. A lexical rule has arguments signifying its length, generally one (`s(0)`), its surface orthographic form, and the DAG representing its feature structure. The DAG is given as a list of feature paths followed by a list of reentrancies.

E.1.1 Adjective

```
Input:  Parsing "a glarf passenger" as a noun phrase
Result:  glarf is an adjective

[[lex_rule([s(0),
             [glarf|_77836],
             [[[maj,adj],
               [pecParseTree,[[daughters,[[word,glarf]]],
                                [rule_number,rn_1001]]],
               [phon,[[eol,_77955],[list,[[first,glarf],[rest,_77955]]]]],
               [synsem,[[local,[[category,
                                   [[bar,zero],
                                    [comps,end],
                                    [head,[[maj,adj],
                                           [mod,
                                            [[synsem,
                                              [[local,
                                                [[content,
                                                  [[index,_78169],
                                                   [restr,
                                                    [[first,_78208],
                                                     [rest,_78237]]]]]]]]]]]]],
                                    [subj,end]]],
                                  [content,[[index,_78169],
                                            [restr,[[first,_78208],
                                                    [rest,[[first,
                                                            [[arg,_78396],
                                                             [relation,_78425]]],
                                                           [rest,_78237]]]]]]]]]]]],
              [[_78169,_78396],
               [_77955,t],
               [_78208,t],
               [_78396,t],
               [_78237,t],
               [_78425,t]]]]),
  [lex_rule([s(0),
             [glarf|_77836],
              [[[maj,adj],
                [pecParseTree,[[daughters,[[word,glarf]]],
                                [rule_number,rn_1001]]],
                [phon,_136802],
                [qstore,_136808],
                [synsem,[[local,[[category,[[bar,zero],
                             [comps,_137054],
                             [head,_136844],
                             [marking,_137066],
                             [spr,_137072],
                             [subj,_136856]]],
                  [conj,nil],
                  [content,_136868],
                  [context,_136874]]]]]],
[[_136194,n],
 [_136200,[[dtrs,[[adjdtr,_136214],[headdtr,_136220]]],
           [maj,_136194],
           [pecParseTree,[[daughters,[[x1,_136214],
                                       [x2,_136220]]],
                           [rule_number,rn_19]]],
           [phon,[[eol,_136268],[list,_136274]]],
           [qstore,[[eol,_136286],[list,_136292]]],
           [synsem,_136298]]],
 [_136220,[[dtrs,[[headdtr,_136316]]],
           [maj,_136194],
           [pecParseTree,[[daughters,[[x1,_136316]]],
                           [rule_number,rn_6]]],
           [phon,_136352],
           [qstore,_136358],
           [synsem,_136364]]],
 [_136358,[[eol,_136286],[list,_136286]]],
 [_136352,[[eol,_136268],[list,_136400]]],
 [_136316,[[maj,_136194],
           [pecParseTree,[[daughters,[[pecPhon,passenger]]],
                           [rule_number,rn_341]]],
           [phon,_136352],
           [qstore,_136358],
           [synsem,[[local,[[category,[[bar,zero],
                                       [comps,_136478],
                                       [head,_136484],
                                       [marking,_136490],
                                       [spr,_136496],
                                       [subj,_136502]]],
                            [conj,nil],
                            [content,_136514],
                            [context,_136520]]]]]],
 [_136496,[[first,[[synsem,_136538]]],[rest,end]]],
 [_136478,end],
 [_136520,[[eol,_136562],[list,_136562]]],
 [_136514,[[index,_136580],
          [restr,[[first,_136592],[rest,_136598]]]]],
[_136580,[[gend,neuter],[num,singular],[per,third]]],
[_136364,[[local,[[category,[[bar,_136646],
                             [comps,_136478],
                             [head,_136484],
                             [marking,_136490],
                             [spr,_136496],
                             [subj,_136502]]],
                  [conj,_136682],
                  [content,_136514],
                  [context,_136520]]]]],
[_136682,nil],
[_136646,one],
[_136598,end],
[_136592,[[instance,_136580],[relation,passenger]]],
[_136400,[[first,passenger],[rest,_136268]]],
[_136214,[[dtrs,[[headdtr,_136766]]],
          [maj,_136772],
          [pecParseTree,[[daughters,[[x1,_136766]]],
                         [rule_number,rn_3]]],
          [phon,_136802],
          [qstore,_136808],
          [synsem,[[local,[[category,[[bar,two],
                                      [comps,end],
                                      [head,_136844],
                                      [spr,end],
                                      [subj,_136856]]],
                           [conj,nil],
                           [content,_136868],
                           [context,_136874]]]]]],
[_136874,[[eol,_136562],[list,_136892]]],
[_136856,end],
[_136844,[[maj,_136772],[mod,[[synsem,_136364]]]]],
[_136808,[[eol,_136286],[list,_136292]]],
[_136802,[[eol,_136400],[list,_136274]]],
[_136772,adj],
[_136766,[[dtrs,[[headdtr,_136982]]],
          [maj,_136772],
          [pecParseTree,[[daughters,[[x1,_136982]]],
                         [rule_number,rn_8]]],
          [phon,_136802],
          [qstore,_136808],
          [synsem,[[local,[[category,[[bar,one],
                                      [comps,_137054],
                                      [head,_136844],
                                      [marking,_137066],
                                      [spr,_137072],
                                      [subj,_136856]]],
                           [conj,nil],
                           [content,_136868],
                           [context,_136874]]]]]],
[_137072,end],
[_137054,end],
[_136982,[[maj,adj],
          [pecParseTree,[[daughters,[[word,glarf]]],
                         [rule_number,rn_1001]]],
          [phon,_136802],
          [qstore,_136808],
          [synsem,[[local,[[category,[[bar,zero],
                                      [comps,_137054],
                                      [head,_136844],
                                      [marking,_137066],
                                      [spr,_137072],
                                      [subj,_136856]]],
                           [conj,nil],
                           [content,_136868],
                           [context,_136874]]]]]],
[_137066,t],
[_136298,[[local,[[category,[[bar,_136646],
                             [comps,_136478],
                             [head,_136484],
                             [marking,_136490],
                             [spr,_136496],
                             [subj,_136502]]],
                  [conj,_136682],
                  [content,_136868],
                  [context,[[eol,_136562],[list,_136892]]]]]]],
[_136892,t],
[_136562,t],
[_136868,[[index,_136580],
          [restr,[[first,_136592],
                  [rest,[[first,[[arg,_136580],
                                 [relation,
                                  disj([accordingto,belligerent,
                                        dead,disinterested,
                                        eastbound,further,
                                        here,just,little,loc,
                                        northbound,of,on,
                                        partitive,sideways,
                                        southbound,speed,still,
                                        supposedly,unable,
                                        westbound,with])]]],
                         [rest,_136598]]]]]]],
[_136502,t],
[_136538,[[local,[[category,[[bar,nil],
                             [head,[[maj,_137472],
                                    [spec,[[synsem,_136298]]]]]]],
                  [conj,nil],
                  [content,_137496],
                  [context,[[eol,_136892],[list,_136892]]]]]]],
[_137472,det],
[_136490,unmarked],
[_136484,[[case,disj([acc,nom])],[form,norm],[maj,_136194]]],
[_136292,t],
[_136286,t],
[_136274,[[first,glarf],[rest,_136400]]],
[_136268,t],
[_137496,[[det,indefinite],[restind,_136868]]],
[_137614,[[maj,_137472],
          [pecParseTree,[[daughters,[[pecPhon,a]]],
                         [rule_number,rn_216]]],
          [phon,[[eol,_136274],[list,_137664]]],
          [qstore,[[eol,_136292],[list,_137682]]],
          [synsem,_136538]]],
[_137682,[[first,_137496],[rest,_136292]]],
[_137664,[[first,a],[rest,_136274]]],
[_136182,[[dtrs,[[headdtr,_136200],
                 [sprdtrs,[[first,_137614],[rest,end]]]]],
          [maj,_136194],
          [pecParseTree,[[daughters,[[x1,_137614],
                                     [x2,_136200]]],
                         [rule_number,rn_5]]],
          [phon,[[eol,_136268],[list,_137664]]],
          [qstore,[[eol,_136286],[list,_137682]]],
          [synsem,[[local,[[category,[[bar,two],
                                      [comps,end],
                                      [head,_136484],
                                      [marking,_136490],
                                      [spr,end],
                                      [subj,_136502]]],
                           [conj,nil],
                           [content,_136868],
                                          [context,[[eol,_136562],
                                                    [list,_136892]]]]]]]]]]]])]]]
```

E.1.2 Adjectival Preposition

```
Input: parsing "glarf elm" as an adjectival phrase
Result:  glarf is a transitive adjective (preposition)

[[lex_rule([s(0),
            [glarf|_2273],
            [[[maj,adj],
              [pecParseTree,[[daughters,[[word,glarf]]],
                             [rule_number,rn_1001]]],
              [phon,[[eol,_2274],[list,[[first,glarf],[rest,_2274]]]]],
              [synsem,[[local,[[category,[[bar,zero],
                                          [head,
                                           [[mod,
                                             [[synsem,
                                               [[local,
                                                 [[content,
                                                   [[index,_2275],
                                                    [restr,
                                                     [[first,_2276],
                                                      [rest,_2277]]]]]]]]]]]]]]],
                               [content,[[index,_2275],
                                         [restr,[[first,_2276],
                                                 [rest,[[first,
                                                         [[arg,_2278],
                                                          [relation,_2279]]],
                                                        [rest,_2277]]]]]]]]]]]],
             [[_2275,_2278],
              [_2274,t],
              [_2276,t],
              [_2278,t],
              [_2277,t],
              [_2279,t]]]]]),
  [lex_rule([s(0),
             [glarf|_2273],
             [[[maj,adj],
               [pecParseTree,[[daughters,[[word,glarf]]],
                              [rule_number,rn_1001]]],
               [phon,[[eol,_41658],[list,_42240]]],
               [qstore,[[eol,_41634],[list,_42258]]],
               [synsem,[[local,[[category,[[bar,zero],
                             [comps,[[first,[[synsem,_41622]]],
                                     [rest,end]]],
                             [head,_42312],
                             [subj,_42318]]],
                  [conj,nil],
                  [content,_42330],
                  [context,[[eol,_41826],[list,_42348]]]]]]]],
[[_41554,adj],
 [_41560,[[dtrs,[[headdtr,_41574]]],
          [maj,_41580],
          [pecParseTree,[[daughters,[[x1,_41574]]],
                         [rule_number,rn_1]]],
          [phon,_41610],
          [qstore,_41616],
          [synsem,_41622]]],
 [_41616,[[eol,_41634],[list,_41634]]],
 [_41610,[[eol,_41652],[list,_41658]]],
 [_41574,[[dtrs,[[headdtr,_41676]]],
          [maj,_41580],
          [pecParseTree,[[daughters,[[x1,_41676]]],
                         [rule_number,rn_6]]],
          [phon,_41610],
          [qstore,_41616],
          [synsem,[[local,[[category,[[bar,one],
                                      [comps,_41748],
                                      [head,_41754],
                                      [marking,_41760],
                                      [spr,_41766],
                                      [subj,_41772]]],
                           [conj,nil],
                           [content,_41784],
                           [context,_41790]]]]]]],
 [_41790,[[backgr,[[first,[[name,elm]]]]],
          [eol,_41820],[list,_41826]]],
 [_41766,end],
 [_41748,end],
 [_41676,[[maj,_41580],
          [pecParseTree,[[daughters,[[pecPhon,elm]]],
                         [rule_number,rn_246]]],
          [phon,_41610],
          [qstore,_41616],
          [synsem,[[local,[[category,[[bar,zero],
                                      [comps,_41748],
                                      [head,_41754],
                                     [marking,_41760],
                                     [spr,_41766],
                                     [subj,_41772]]],
                          [conj,nil],
                          [content,_41784],
                          [context,_41790]]]]]],
[_41760,unmarked],
[_41784,[[index,_41976]]],
[_41976,[[gend,neuter],[num,singular],[per,third]]],
[_41772,t],
[_41754,[[case,disj([acc,nom])],[form,norm],[maj,_41580]]],
[_41580,n],
[_41622,[[local,[[category,[[bar,two],
                            [comps,end],
                            [head,_41754],
                            [spr,end],
                            [subj,_41772]]],
                 [conj,nil],
                 [content,_41784],
                 [context,_41790]]]]],
[_41826,[[first,[[backgr,[[first,[[bearer,_41976],
                                  [relation,naming]]]]]],
                          [rest,_41820]]],
[_41820,t],
[_41634,t],
[_41658,[[first,Elm St],[rest,_41652]]],
[_41652,t],
[_42190,[[maj,adj],
         [pecParseTree,[[daughters,[[word,glarf]]],
                        [rule_number,rn_1001]]],
         [phon,[[eol,_41658],[list,_42240]]],
         [qstore,[[eol,_41634],[list,_42258]]],
         [synsem,[[local,[[category,[[bar,zero],
                                     [comps,[[first,
                                              [[synsem,_41622]]],
                                             [rest,end]]],
                                     [head,_42312],
                                     [subj,_42318]]],
                          [conj,nil],
                          [content,_42330],
                          [context,[[eol,_41826],
                                    [list,_42348]]]]]]]],
[_42330,[[index,_42360],
         [restr,[[first,_42372],
                             [rest,[[first,[[arg,_42360],
                                           [relation,disj([across,along,
                                                           approach,at,
                                                           cover,cross,
                                                           down,from,
                                                           hasproperty,
                                                           in,meet,of,
                                                           onto,partitive,
                                                           path_change,
                                                           street,takepath,
                                                           through])]]],
                                    [rest,_42440]]]]]]],
            [_42372,t],
            [_42360,t],
            [_42440,t],
            [_42312,[[maj,_41554],
                     [mod,[[synsem,
                            [[local,
                              [[content,[[index,_42360],
                                         [restr,[[first,_42372],
                                                 [rest,_42440]]]]]]]]]]]],
            [_42240,[[first,glarf],[rest,_41658]]],
            [_42318,t],
            [_42348,t],
            [_42258,t],
            [_41542,[[dtrs,[[compdtrs,[[first,_41560],[rest,end]]],
                            [headdtr,_42190]]],
                     [maj,_41554],
                     [pecParseTree,[[daughters,[[x1,_42190],[x2,_41560]]],
                                    [rule_number,rn_17]]],
                     [phon,[[eol,_41652],[list,_42240]]],
                     [qstore,[[eol,_41634],[list,_42258]]],
                     [synsem,[[local,[[category,[[bar,one],
                                                 [comps,end],
                                                 [head,_42312],
                                                 [subj,_42318]]],
                                      [conj,nil],
                                      [content,_42330],
                                      [context,[[eol,_41820],
                                               [list,_42348]]]]]]]]]]]),
lex_rule([s(0),
          [glarf|_2273],
          [[[maj,adj],
            [pecParseTree,[[daughters,[[word,glarf]]],
                [rule_number,rn_1001]]],
 [phon,[[eol,_51244],[list,_51700]]],
 [qstore,[[eol,_51220],[list,_51718]]],
 [synsem,[[local,[[category,[[bar,zero],
                             [comps,[[first,[[synsem,_51208]]],
                                     [rest,end]]],
                             [head,_51772],
                             [subj,_51778]]],
                  [conj,nil],
                  [content,_51790],
                  [context,[[eol,_51400],[list,_51808]]]]]]]],
[[_51140,adj],
 [_51146,[[dtrs,[[headdtr,_51160]]],
          [maj,_51166],
          [pecParseTree,[[daughters,[[x1,_51160]]],
                         [rule_number,rn_6]]],
          [phon,_51196],
          [qstore,_51202],
          [synsem,_51208]]],
 [_51202,[[eol,_51220],[list,_51220]]],
 [_51196,[[eol,_51238],[list,_51244]]],
 [_51160,[[maj,_51166],
          [pecParseTree,[[daughters,[[pecPhon,elm]]],
                         [rule_number,rn_246]]],
          [phon,_51196],
          [qstore,_51202],
          [synsem,[[local,[[category,[[bar,zero],
                                      [comps,_51322],
                                      [head,_51328],
                                      [marking,_51334],
                                      [spr,_51340],
                                      [subj,_51346]]],
                           [conj,nil],
                           [content,_51358],
                           [context,_51364]]]]]]],
 [_51364,[[backgr,[[first,[[name,elm]]]]],
          [eol,_51394],
          [list,_51400]]],
 [_51358,[[index,_51412]]],
 [_51412,[[gend,neuter],[num,singular],[per,third]]],
 [_51340,end],
 [_51334,unmarked],
 [_51328,[[case,disj([acc,nom])],[form,norm],[maj,_51166]]],
 [_51166,n],
[_51322,end],
[_51346,t],
[_51208,[[local,[[category,[[bar,one],
                            [comps,_51322],
                            [head,_51328],
                            [marking,_51334],
                            [spr,_51340],
                            [subj,_51346]]],
                 [conj,nil],
                 [content,_51358],
                 [context,_51364]]]]],
[_51400,[[first,[[backgr,[[first,[[bearer,_51412],
                                  [relation,naming]]]]]]],
         [rest,_51394]]],
[_51394,t],
[_51220,t],
[_51244,[[first,Elm St],[rest,_51238]]],
[_51238,t],
[_51650,[[maj,adj],
         [pecParseTree,[[daughters,[[word,glarf]]],
                        [rule_number,rn_1001]]],
         [phon,[[eol,_51244],[list,_51700]]],
         [qstore,[[eol,_51220],[list,_51718]]],
         [synsem,[[local,[[category,[[bar,zero],
                                     [comps,[[first,
                                              [[synsem,_51208]]],
                                             [rest,end]]],
                                     [head,_51772],
                                     [subj,_51778]]],
                          [conj,nil],
                          [content,_51790],
                          [context,[[eol,_51400],
                                    [list,_51808]]]]]]]],
[_51790,[[index,_51820],
         [restr,[[first,_51832],
                 [rest,[[first,[[arg,_51820],
                                [relation,disj([across,along,
                                                approach,at,
                                                cover,cross,
                                                down,from,
                                                hasproperty,
                                                in,meet,of,
                                                onto,partitive,
                                                path_change,
                                                            street,takepath,
                                                            through])]]],
                                    [rest,_51900]]]]]]],
            [_51832,t],
            [_51820,t],
            [_51900,t],
            [_51772,[[maj,_51140],
                     [mod,[[synsem,
                            [[local,
                              [[content,
                                [[index,_51820],
                                 [restr,[[first,_51832],
                                         [rest,_51900]]]]]]]]]]]]],
            [_51700,[[first,glarf],[rest,_51244]]],
            [_51778,t],
            [_51808,t],
            [_51718,t],
            [_51128,[[dtrs,[[compdtrs,[[first,_51146],[rest,end]]],
                            [headdtr,_51650]]],
                     [maj,_51140],
                     [pecParseTree,[[daughters,[[x1,_51650],[x2,_51146]]],
                                    [rule_number,rn_17]]],
                     [phon,[[eol,_51238],[list,_51700]]],
                     [qstore,[[eol,_51220],[list,_51718]]],
                     [synsem,[[local,[[category,[[bar,one],
                                                 [comps,end],
                                                 [head,_51772],
                                                 [subj,_51778]]],
                                      [conj,nil],
                                      [content,_51790],
                                      [context,[[eol,_51394],
                                                [list,_51808]]]]]]]]]]]]),

lex_rule([s(0),
          [glarf|_2273],
          [[[maj,adj],
            [pecParseTree,[[daughters,[[word,glarf]]],
                           [rule_number,rn_1001]]],
            [phon,[[eol,_59652],[list,_59964]]],
            [qstore,[[eol,_59664],[list,_59982]]],
            [synsem,[[local,[[category,[[bar,zero],
                                        [comps,[[first,[[synsem,_59676]]],
                                                [rest,end]]],
                                        [head,_60036],
                                        [subj,_60042]]],
                  [conj,nil],
                  [content,_60054],
                  [context,[[eol,_59808],[list,_60072]]]]]]]],
[[_59596,adj],
 [_59602,[[maj,_59610],
          [pecParseTree,[[daughters,[[pecPhon,elm]]],
                         [rule_number,rn_246]]],
          [phon,[[eol,_59646],[list,_59652]]],
          [qstore,[[eol,_59664],[list,_59664]]],
          [synsem,_59676]]],
 [_59610,n],
 [_59676,[[local,[[category,[[bar,zero],
                             [comps,end],
                             [head,[[case,disj([acc,nom])],
                                    [form,norm],
                                    [maj,_59610]]],
                             [marking,unmarked],
                             [spr,end]]],
                  [conj,nil],
                  [content,[[index,_59772]]],
                  [context,[[backgr,[[first,[[name,elm]]]]],
                            [eol,_59802],
                            [list,_59808]]]]]]],
 [_59772,[[gend,neuter],[num,singular],[per,third]]],
 [_59808,[[first,[[backgr,[[first,[[bearer,_59772],
                                   [relation,naming]]]]]]],
          [rest,_59802]]],
 [_59802,t],
 [_59664,t],
 [_59652,[[first,Elm St],[rest,_59646]]],
 [_59646,t],
 [_59914,[[maj,adj],
          [pecParseTree,[[daughters,[[word,glarf]]],
                         [rule_number,rn_1001]]],
          [phon,[[eol,_59652],[list,_59964]]],
          [qstore,[[eol,_59664],[list,_59982]]],
          [synsem,[[local,[[category,[[bar,zero],
                                      [comps,[[first,
                                              [[synsem,_59676]]],
                                             [rest,end]]],
                                      [head,_60036],
                                      [subj,_60042]]],
                           [conj,nil],
                           [content,_60054],
                          [context,[[eol,_59808],
                                    [list,_60072]]]]]]]]],
[_60054,[[index,_60084],
         [restr,[[first,_60096],
                 [rest,[[first,[[arg,_60084],
                                [relation,disj([across,along,
                                                approach,at,
                                                cover,cross,
                                                down,from,
                                                hasproperty,
                                                in,meet,of,
                                                onto,partitive,
                                                path_change,
                                                street,takepath,
                                                through])]]],
                        [rest,_60164]]]]]]],
[_60096,t],
[_60084,t],
[_60164,t],
[_60036,[[maj,_59596],
         [mod,[[synsem,[[local,
                         [[content,
                           [[index,_60084],
                            [restr,[[first,_60096],
                                    [rest,_60164]]]]]]]]]]]],
[_59964,[[first,glarf],[rest,_59652]]],
[_60042,t],
[_60072,t],
[_59982,t],
[_59584,[[dtrs,[[compdtrs,[[first,_59602],
                           [rest,end]]],
                [headdtr,_59914]]],
         [maj,_59596],
         [pecParseTree,[[daughters,[[x1,_59914],[x2,_59602]]],
                        [rule_number,rn_17]]],
         [phon,[[eol,_59646],[list,_59964]]],
         [qstore,[[eol,_59664],[list,_59982]]],
         [synsem,[[local,[[category,[[bar,one],
                                     [comps,end],
                                     [head,_60036],
                                     [subj,_60042]]],
                          [conj,nil],
                          [content,_60054],
                          [context,[[eol,_59802],
                                                    [list,_60072]]]]]]]]]]]])]]]
```

E.1.3 Adverb

```
Input:  Parsing "Meuser glarf followed" as a sentence
Result:  glarf is an adverb

[[lex_rule([s(0),
            [glarf|_152513],
            [[[maj,adv],
              [pecParseTree,[[daughters,[[word,glarf]]],[rule_number,rn_1001]]],
              [phon,[[eol,_152514],[list,[[first,glarf],[rest,_152514]]]]],
              [synsem,[[local,[[category,[[bar,zero],
                                           [comps,end],
                                           [head,[[maj,adv],
                                                  [mod,
                                                   [[synsem,
                                                     [[local,
                                                       [[content,
                                                         [[psoa,_152515],
                                                          [restr,_152516]]]]]]]]]]],
                                           [subj,t]]],
                                 [content,[[psoa,_152515],
                                           [restr,[[first,[[relation,_152517],
                                                            [soaarg,_152518]]],
                                                   [rest,_152516]]]]]]]]],
             [[_152515,_152518],
              [_152514,t],
              [_152518,t],
              [_152516,t],
              [_152517,t]]]]),
  [lex_rule([s(0),
             [glarf|_152513],
             [[[maj,adv],
               [pecParseTree,[[daughters,[[word,glarf]]],
                              [rule_number,rn_1001]]],
               [phon,_225011],
               [qstore,_225017],
               [synsem,[[local,[[category,[[bar,zero],
                                            [comps,_225257],
                                            [head,_225053],
                                            [marking,_225269],
                                            [spr,_225275],
                             [subj,_225065]]],
                  [conj,nil],
                  [content,_224321],
                  [context,_225083]]]]]],
[[_224181,[[form,finite],[maj,_224195]]],
 [_224195,v],
 [_224205,[[dtrs,[[headdtr,_224219]]],
           [maj,_224195],
           [pecParseTree,[[daughters,[[x1,_224219]]],
                          [rule_number,rn_2]]],
           [phon,_224255],
           [qstore,_224261],
           [synsem,[[local,[[category,[[bar,two],
                                       [comps,end],
                                       [head,_224181],
                                       [spr,end],
                                       [subj,_224309]]],
                            [conj,nil],
                            [content,_224321],
                            [context,_224327]]]]]]],
 [_224327,[[eol,_224339],[list,_224345]]],
 [_224309,[[first,[[synsem,_224363]]],[rest,end]]],
 [_224261,[[eol,_224381],[list,_224387]]],
 [_224255,[[eol,_224399],[list,_224405]]],
 [_224219,[[dtrs,[[adjdtr,_224423],[headdtr,_224429]]],
           [maj,_224195],
           [pecParseTree,[[daughters,[[x1,_224423],[x2,_224429]]],
                          [rule_number,rn_22]]],
           [phon,_224255],
           [qstore,_224261],
           [synsem,[[local,[[category,[[bar,_224501],
                                       [comps,_224507],
                                       [head,_224181],
                                       [marking,_224519],
                                       [spr,_224525],
                                       [subj,_224309]]],
                            [conj,_224537],
                            [content,_224321],
                            [context,_224327]]]]]]],
 [_224537,nil],
 [_224525,end],
 [_224507,end],
 [_224501,one],
 [_224429,[[dtrs,[[headdtr,_224591]]],
          [maj,_224195],
          [pecParseTree,[[daughters,[[x1,_224591]]],
                         [rule_number,rn_7]]],
          [phon,_224627],
          [qstore,_224633],
          [synsem,_224639]]],
[_224633,[[eol,_224381],[list,_224381]]],
[_224627,[[eol,_224399],[list,_224675]]],
[_224591,[[maj,_224195],
          [pecParseTree,[[daughters,[[pecPhon,followed]]],
                         [rule_number,rn_487]]],
          [phon,_224627],
          [qstore,_224633],
          [synsem,[[local,[[category,[[bar,zero],
                                      [comps,_224507],
                                      [head,_224181],
                                      [marking,_224519],
                                      [spr,_224525],
                                      [subj,_224309]]],
                           [conj,nil],
                           [content,_224789],
                           [context,_224795]]]]]]],
[_224795,[[eol,_224339],[list,_224339]]],
[_224789,[[psoa,_224825],[restr,_224831],[tense,past]]],
[_224639,[[local,[[category,[[bar,_224501],
                             [comps,_224507],
                             [head,_224181],
                             [marking,_224519],
                             [spr,_224525],
                             [subj,_224309]]],
                  [conj,_224537],
                  [content,_224789],
                  [context,_224795]]]]],
[_224831,end],
[_224825,[[follower,_224927],[relation,follow]]],
[_224519,t],
[_224675,[[first,followed],[rest,_224399]]],
[_224423,[[dtrs,[[headdtr,_224975]]],
          [maj,_224981],
          [pecParseTree,[[daughters,[[x1,_224975]]],
                         [rule_number,rn_4]]],
          [phon,_225011],
          [qstore,_225017],
          [synsem,[[local,[[category,[[bar,two],
                                      [comps,end],
                                      [head,_225053],
                                      [spr,end],
                                      [subj,_225065]]],
                           [conj,nil],
                           [content,_224321],
                           [context,_225083]]]]]],
[_225083,[[eol,_224339],[list,_224345]]],
[_225053,[[maj,_224981],[mod,[[synsem,_224639]]]]],
[_225017,[[eol,_224381],[list,_224387]]],
[_225011,[[eol,_224675],[list,_224405]]],
[_224981,adv],
[_224975,[[dtrs,[[headdtr,_225185]]],
          [maj,_224981],
          [pecParseTree,[[daughters,[[x1,_225185]]],
                         [rule_number,rn_9]]],
          [phon,_225011],
          [qstore,_225017],
          [synsem,[[local,[[category,[[bar,one],
                                      [comps,_225257],
                                      [head,_225053],
                                      [marking,_225269],
                                      [spr,_225275],
                                      [subj,_225065]]],
                           [conj,nil],
                           [content,_224321],
                           [context,_225083]]]]]],
[_225275,end],
[_225257,end],
[_225185,[[maj,adv],
          [pecParseTree,[[daughters,[[word,glarf]]],
                         [rule_number,rn_1001]]],
          [phon,_225011],
          [qstore,_225017],
          [synsem,[[local,[[category,[[bar,zero],
                                      [comps,_225257],
                                      [head,_225053],
                                      [marking,_225269],
                                      [spr,_225275],
                                      [subj,_225065]]],
                           [conj,nil],
                           [content,_224321],
                           [context,_225083]]]]]],
[_225269,t],
[_225065,t],
[_224345,t],
[_224339,t],
[_224321,[[psoa,_224825],
          [restr,[[first,[[relation,disj([accordingto,further,
                                          here,just,little,loc,
                                          of,on,sideways,still,
                                          supposedly,with])],
                          [soaarg,_224825]]],
                  [rest,_224831]]]]],
[_224363,[[local,[[category,[[bar,two],
                             [comps,end],
                             [head,_225559],
                             [spr,end],
                             [subj,_225571]]],
                  [conj,nil],
                  [content,_224927],
                  [context,_225589]]]]],
[_224927,[[index,_225601]]],
[_225571,end],
[_225559,[[case,nom],[form,norm],[maj,_225631]]],
[_225631,n],
[_224387,t],
[_224381,t],
[_224405,[[first,glarf],[rest,_224675]]],
[_224399,t],
[_225589,[[backgr,[[first,[[name,meuser]]]]],
          [eol,_224345],
          [list,_225709]]],
[_225601,[[gend,neuter],[num,singular],[per,third]]],
[_225737,[[dtrs,[[headdtr,_225751]]],
          [maj,_225631],
          [pecParseTree,[[daughters,[[x1,_225751]]],
                         [rule_number,rn_1]]],
          [phon,_225787],
          [qstore,_225793],
          [synsem,_224363]]],
[_225793,[[eol,_224387],[list,_224387]]],
[_225787,[[eol,_224405],[list,_225835]]],
[_225751,[[dtrs,[[headdtr,_225853]]],
          [maj,_225631],
          [pecParseTree,[[daughters,[[x1,_225853]]],
                         [rule_number,rn_6]]],
          [phon,_225787],
          [qstore,_225793],
          [synsem,[[local,[[category,[[bar,one],
                                      [comps,_225925],
                                      [head,_225559],
                                      [marking,_225937],
                                      [spr,_225943],
                                      [subj,_225571]]],
                           [conj,nil],
                           [content,_224927],
                           [context,_225589]]]]]],
[_225943,end],
[_225925,end],
[_225853,[[maj,_225631],
          [pecParseTree,[[daughters,[[pecPhon,meuser]]],
                         [rule_number,rn_242]]],
          [phon,_225787],
          [qstore,_225793],
          [synsem,[[local,[[category,[[bar,zero],
                                      [comps,_225925],
                                      [head,_225559],
                                      [marking,_225937],
                                      [spr,_225943],
                                      [subj,_225571]]],
                           [conj,nil],
                           [content,_224927],
                           [context,_225589]]]]]],
[_225937,unmarked],
[_225709,[[first,[[backgr,[[first,[[bearer,_225601],
                                   [relation,naming]]]]]]],
          [rest,_224345]]],
[_225835,[[first,meuser],[rest,_224405]]],
[_224169,[[dtrs,[[headdtr,_224205],
                 [subjdtrs,[[first,_225737],[rest,end]]]]],
          [maj,_224195],
          [pecParseTree,[[daughters,[[x1,_225737],[x2,_224205]]],
                         [rule_number,rn_0]]],
          [phon,[[eol,_224399],[list,_225835]]],
          [qstore,[[eol,_224381],[list,_224387]]],
          [synsem,[[local,[[category,[[bar,two],
                                      [comps,end],
                                      [head,_224181],
                                      [spr,end],
                                      [subj,end]]],
                           [conj,nil],
                                         [content,_224321],
                                         [contert,[[eol,_224339],
                                                   [list,_225709]]]]]]]]]]])]]]
```

E.1.4 Determiner

```
Input: after glarf accident, parsed as an adjective phrase
Result:  glarf is a determiner

[[glarf,[[lex_rule([s(0),
                   [glarf|_42629],
                   [[[maj,det],
                     [pecParseTree,[[daughters,[[word,glarf]]],
                                    [rule_number,rn_1001]]],
                     [phon,[[eol,_42746],[list,[[first,glarf],[rest,_42746]]]]],
                     [synsem,[[local,[[category,
                                       [[bar,zero],
                                        [head,
                                         [[spec,
                                           [[synsem,
                                             [[local,
                                               [[content,_42914]]]]]]]]]]],
                                      [content,[[det,_42971],
                                                [restind,_42914]]]]]]]],
                    [[_42746,t],[_42914,t],[_42971,t]]]]),
          [lex_rule([s(0),
                    [glarf|_42629],
                    [[[maj,det],
                      [pecParseTree,[[daughters,[[word,glarf]]],
                                     [rule_number,rn_1001]]],
                      [phon,[[eol,_102552],[list,_102420]]],
                      [qstore,[[eol,_102432],[list,_102438]]],
                      [synsem,_102732]],
                     [[_102328,adj],
                      [_102334,[[dtrs,[[headdtr,_102348],
                                       [sprdtrs,[[first,_102360],
                                                 [rest,end]]]]],
                                [maj,_102372],
                                [pecParseTree,[[daughters,[[x1,_102360],
                                                           [x2,_102348]]],
                                               [rule_number,rn_5]]],
                                [phon,[[eol,_102414],[list,_102420]]],
                                [qstore,[[eol,_102432],[list,_102438]]],
          [synsem,_102444]]],
[_102372,n],
[_102348,[[dtrs,[[headdtr,_102468]]],
          [maj,_102372],
          [pecParseTree,[[daughters,[[x1,_102468]]],
                         [rule_number,rn_6]]],
          [phon,_102504],
          [qstore,_102510],
          [synsem,_102516]]],
[_102510,[[eol,_102432],[list,_102432]]],
[_102504,[[eol,_102414],[list,_102552]]],
[_102468,[[maj,_102372],
          [pecParseTree,[[daughters,[[pecPhon,accident]]],
                         [rule_number,rn_255]]],
          [phon,_102504],
          [qstore,_102510],
          [synsem,[[local,[[category,[[bar,zero],
                                      [comps,_102630],
                                      [head,_102636],
                                      [marking,_102642],
                                      [spr,_102648],
                                      [subj,_102654]]],
                           [conj,nil],
                           [content,_102666],
                           [context,_102672]]]]]]],
[_102636,[[case,acc],[form,norm],[maj,_102372]]],
[_102672,[[eol,_102708],[list,_102708]]],
[_102648,[[first,[[synsem,_102732]]],[rest,end]]],
[_102630,end],
[_102516,[[local,[[category,[[bar,one],
                             [comps,_102630],
                             [head,_102636],
                             [marking,_102642],
                             [spr,_102648],
                             [subj,_102654]]],
                  [conj,nil],
                  [content,_102666],
                  [context,_102672]]]]],
[_102654,t],
[_102732,[[local,[[category,[[bar,zero],
                             [head,
                              [[maj,det],
                               [spec,
                                [[synsem,_102516]]]]]]],
                  [content,[[det,_102882],
                            [restind,_102666]]],
                  [context,[[eol,_102708],
                            [list,_102906]]]]]],
[_102642,unmarked],
[_102552,[[first,accident],[rest,_102414]]],
[_102360,[[maj,det],
          [pecParseTree,[[daughters,[[word,glarf]]],
                         [rule_number,rn_1001]]],
          [phon,[[eol,_102552],[list,_102420]]],
          [qstore,[[eol,_102432],[list,_102438]]],
          [synsem,_102732]]],
[_102444,[[local,[[category,[[bar,two],
                             [comps,end],
                             [head,_102636],
                             [marking,_102642],
                             [spr,end],
                             [subj,_102654]]],
                  [conj,nil],
                  [content,_102666],
                  [context,[[eol,_102708],
                            [list,_102906]]]]]],
[_102906,t],
[_102708,t],
[_102666,[[index,_103116],
          [restr,[[first,[[instance,_103116],
                          [relation,accident]]],
                  [rest,end]]]]],
[_103116,[[gend,neuter],[num,singular],[per,third]]],
[_102438,t],
[_102432,t],
[_102420,[[first,glarf],[rest,_102552]]],
[_102414,t],
[_103210,[[maj,_102328],
          [pecParseTree,[[daughters,[[pecPhon,after]]],
                         [rule_number,rn_73]]],
          [phon,[[eol,_102420],[list,_103260]]],
          [qstore,[[eol,_102438],[list,_102438]]],
          [synsem,[[local,[[category,
                            [[bar,zero],
                             [comps,[[first,
                                      [[synsem,_102444]]],
                                     [rest,end]]],
                             [head,_103332],
                             [subj,_103338]]],
                           [conj,nil],
                           [content,_103350],
                           [context,
                            [[eol,_102906],
                             [list,_102906]]]]]]]],
[_103350,[[index,_103380],
          [restr,[[first,_103392],
                  [rest,[[first,[[arg,_103410],
                                 [comparator,_102666],
                                 [relation,after]]],
                         [rest,_103428]]]]]],
[_103380,t],
[_103392,t],
[_103428,t],
[_103410,[[index,_103380],
          [restr,[[first,_103392],[rest,_103428]]]]],
[_103338,[[first,[[synsem,_103494]]]]],
[_103494,[[local,[[category,[[head,[[maj,n]]]]],
                  [content,_103410]]]]],
[_103332,[[maj,_102328],[mod,[[synsem,_103494]]]]],
[_103260,[[first,after],[rest,_102420]]],
[_102316,[[dtrs,[[compdtrs,[[first,_102334],
                            [rest,end]]],
                 [headdtr,_103210]]],
          [maj,_102328],
          [pecParseTree,[[daughters,[[x1,_103210],
                                     [x2,_102334]]],
                         [rule_number,rn_17]]],
          [phon,[[eol,_102414],[list,_103260]]],
          [qstore,[[eol,_102432],[list,_102438]]],
          [synsem,[[local,[[category,[[bar,one],
                                      [comps,end],
                                      [head,_103332],
                                      [subj,_103338]]],
                           [conj,nil],
                           [content,_103350],
                           [context,
                            [[eol,_102708],
                             [list,_102906]]]]]]]]]]])]]]]].
```

E.1.5 Noun

```
Input:  Parsing "across glarf" as an adjective phrase
Result:  glarf is a noun

[[glarf,[[lex_rule([s(0),
                    [glarf|_92006],
                    [[[maj,n],
                      [pecParseTree,[[daughters,[[word,glarf]]],
                                     [rule_number,rn_1001]]],
                      [phon,[[eol,_92125],[list,[[first,glarf],[rest,_92125]]]]],
                      [synsem,[[local,[[category,[[bar,zero],
                                                  [comps,end],
                                                  [head,[[case,acc],[maj,n]]],
                                                  [subj,t]]],
                                       [content,[[index,_92349],
                                                 [restr,
                                                  [[first,
                                                    [[instance,_92349],
                                                     [relation,_92421]]]]]]]]]]]],
                     [[_92125,t],[_92349,t],[_92421,t]]]]),
          [lex_rule([s(0),
                     [glarf|_92006],
                     [[[maj,n],
                       [pecParseTree,[[daughters,[[word,glarf]]],
                                      [rule_number,rn_1001]]],
                       [phon,_128253],
                       [qstore,_128259],
                       [synsem,[[local,[[category,[[bar,zero],
                                                   [comps,_128397],
                                                   [head,_128403],
                                                   [marking,_128409],
                                                   [spr,_128415],
                                                   [subj,_128421]]],
                                        [conj,nil],
                                        [content,_128433],
                                        [context,_128439]]]]]],
                      [[_128197,adj],
                       [_128203,[[dtrs,[[headdtr,_128217]]],
                                 [maj,_128223],
                                 [pecParseTree,[[daughters,[[x1,_128217]]],
                                                [rule_number,rn_1]]],
                                 [phon,_128253],
                                 [qstore,_128259],
          [synsem,_128265]]],
[_128223,n],
[_128259,[[eol,_128283],[list,_128289]]],
[_128253,[[eol,_128301],[list,_128307]]],
[_128217,[[dtrs,[[headdtr,_128325]]],
          [maj,_128223],
          [pecParseTree,[[daughters,[[x1,_128325]]],
                         [rule_number,rn_6]]],
          [phon,_128253],
          [qstore,_128259],
          [synsem,[[local,[[category,[[bar,one],
                                      [comps,_128397],
                                      [head,_128403],
                                      [marking,_128409],
                                      [spr,_128415],
                                      [subj,_128421]]],
                           [conj,nil],
                           [content,_128433],
                           [context,_128439]]]]]]],
[_128439,[[eol,_128451],[list,_128457]]],
[_128403,[[case,acc],[form,norm],[maj,_128223]]],
[_128415,end],
[_128397,end],
[_128325,[[maj,n],
          [pecParseTree,[[daughters,[[word,glarf]]],
                         [rule_number,rn_1001]]],
          [phon,_128253],
          [qstore,_128259],
          [synsem,[[local,[[category,[[bar,zero],
                                      [comps,_128397],
                                      [head,_128403],
                                      [marking,_128409],
                                      [spr,_128415],
                                      [subj,_128421]]],
                           [conj,nil],
                           [content,_128433],
                           [context,_128439]]]]]]],
[_128409,t],
[_128421,t],
[_128265,[[local,[[category,[[bar,two],
                            [comps,end],
                            [head,_128403],
                            [spr,end],
                            [subj,_128421]]],
                 [conj,nil],
                 [content,_128433],
                 [context,_128439]]]]],
[_128433,[[index,_128703],
         [restr,
          [[first,
            [[instance,_128703],
             [relation,
              disj([arbordr,barkwoodrd,brushrd,
                    busey,california,centerdr,clay,
                    claystreet,communicationsdr,
                    creekbendrd,crookedcreekrd,
                    cunninghamavenue,curb,
                    devonshireroad,duncanroad,
                    ealgonquinrd,egolfrd,elm,
                    estatepky,fifth,fifthstreet,
                    fourth,fourthstreet,green,
                    gregory,harvey,i74,illinois,
                    interstate57,kirby,lane,lincoln,
                    middledr,motorola,motorolaacc,
                    neil,nevada,nmeachamrd,
                    nplumgroverd,nstatepky,nwestdr,
                    oldefarmrd,oregon,parkingacc,
                    paynerd,pinevalleydr,remingtonrd,
                    road,smeachamrd,southdr,
                    springfield,street,talltimbersrd,
                    towercircledr,towerrd,university,
                    universityavenue,unnamed,
                    vermillionst,vine,vinestreet,
                    westdr,wileyrd])]]]]]]],
[_128703,t],
[_128307,[[first,glarf],[rest,_128301]]],
[_128301,t],
[_128889,[[maj,_128197],
         [pecParseTree,[[daughters,[[pecPhon,across]]],
                       [rule_number,rn_72]]],
         [phon,[[eol,_128307],[list,_128939]]],
         [qstore,[[eol,_128289],[list,_128289]]],
         [synsem,[[local,[[category,
                          [[bar,zero],
                           [comps,
                            [[first,[[synsem,_128265]]],
                             [rest,end]]],
                           [head,_129011],
                             [subj,_129017]]],
                           [conj,nil],
                           [content,_129029],
                           [context,[[eol,_128457],
                                     [list,_128457]]]]]]]]],
[_128457,t],
[_129029,[[index,_129065],
          [restr,[[first,_129077],
                  [rest,[[first,[[arg,_129095],
                                 [comparator,_128433],
                                 [relation,across]]],
                         [rest,_129113]]]]]],
[_129065,t],
[_129077,t],
[_129113,t],
[_129095,[[index,_129065],[restr,[[first,_129077],
                                  [rest,_129113]]]]],
[_129017,[[first,[[synsem,_129179]]]]],
[_129179,[[local,[[category,[[head,[[maj,n]]]]],
                  [content,_129095]]]]],
[_129011,[[maj,_128197],[mod,[[synsem,_129179]]]]],
[_128289,t],
[_128939,[[first,across],[rest,_128307]]],
[_128451,t],
[_128283,t],
[_128185,[[dtrs,[[compdtrs,[[first,_128203],[rest,end]]],
                 [headdtr,_128889]]],
          [maj,_128197],
          [pecParseTree,[[daughters,[[x1,_128889],
                                     [x2,_128203]]],
                         [rule_number,rn_17]]],
          [phon,[[eol,_128301],[list,_128939]]],
          [qstore,[[eol,_128283],[list,_128289]]],
          [synsem,[[local,[[category,[[bar,one],
                                      [comps,end],
                                      [head,_129011],
                                      [subj,_129017]]],
                           [conj,nil],
                           [content,_129029],
                           [context,
                            [[eol,_128451],
                             [list,_128457]]]]]]]]]]])]]]]].
```

E.1.6 Intransitive Verb

```
Input:  Parsing "the driver glarf" as a sentence
Result:  glarf is an intransitive verb

[[lex_rule([s(0),
            [glarf|_48199],
            [[[maj,v],
              [pecParseTree,[[daughters,[[word,glarf]]],[rule_number,rn_1001]]],
              [phon,[[eol,_48200],[list,[[first,glarf],[rest,_48200]]]]],
              [synsem,[[local,[[category,[[bar,zero],
                                            [comps,end],
                                            [head,[[maj,v]]],
                                            [subj,[]]]],
                               [content,[[psoa,[[relation,_48201]]]]]]]]]]],
             [[_48200,t],[_48201,t]]]]),
  [lex_rule([s(0),
             [glarf|_48199],
             [[[maj,v],
               [pecParseTree,[[daughters,[[word,glarf]]],[rule_number,rn_1001]]],
               [phon,_64370],[qstore,_64376],
               [synsem,[[local,[[category,[[bar,zero],
                                             [comps,_64610],
                                             [head,_64296],
                                             [marking,_64622],
                                             [spr,_64628],
                                             [subj,_64424]]],
                                [conj,nil],
                                [content,_64436],
                                [context,_64442]]]]]],
              [[_64296,[[form,finite],[maj,_64310]]],
               [_64310,v],
               [_64320,[[dtrs,[[headdtr,_64334]]],
                        [maj,_64310],
                        [pecParseTree,[[daughters,[[x1,_64334]]],
                                       [rule_number,rn_2]]],
                        [phon,_64370],[qstore,_64376],
                        [synsem,[[local,[[category,[[bar,two],
                                                     [comps,end],
                                                     [head,_64296],
                                                     [spr,end],
                                                     [subj,_64424]]],
                                        [conj,nil],
                                        [content,_64436],
                          [context,_64442]]]]]]],
[_64442,[[eol,_64454],[list,_64460]]],
[_64424,[[first,[[synsem,_64478]]],[rest,end]]],
[_64376,[[eol,_64496],[list,_64502]]],
[_64370,[[eol,_64514],[list,_64520]]],
[_64334,[[dtrs,[[headdtr,_64538]]],
         [maj,_64310],
         [pecParseTree,[[daughters,[[x1,_64538]]],
                        [rule_number,rn_7]]],
         [phon,_64370],
         [qstore,_64376],
         [synsem,[[local,[[category,[[bar,one],
                                     [comps,_64610],
                                     [head,_64296],
                                     [marking,_64622],
                                     [spr,_64628],
                                     [subj,_64424]]],
                          [conj,nil],
                          [content,_64436],
                          [context,_64442]]]]]]],
[_64628,end],
[_64610,end],
[_64538,[[maj,v],
         [pecParseTree,[[daughters,[[word,glarf]]],
                        [rule_number,rn_1001]]],
         [phon,_64370],
         [qstore,_64376],
         [synsem,[[local,[[category,[[bar,zero],
                                     [comps,_64610],
                                     [head,_64296],
                                     [marking,_64622],
                                     [spr,_64628],
                                     [subj,_64424]]],
                          [conj,nil],
                          [content,_64436],
                          [context,_64442]]]]]]],
[_64622,t],
[_64436,[[psoa,[[relation,disj([accordingto,admit,approach,
                                arrive,avoid,back,bear,beat,
                                become,behind,brace,break,by,
                                carry,cause,clear,collide,
                                come,confirm,continue,cross,
                                damage,disobey,dodge,drive,
                                driver,enter,exit,fail,feel,
                                 follow,give,go,halt,
                                 hasproperty,have,head,hear,
                                 hit,injure,inspect,kill,know,
                                 leave,look,lose,need,of,own,
                                 park,partitive,post,proceed,
                                 pronounce,pull,ram,reach,
                                 realize,release,report,
                                 reroute,run,runover,say,see,
                                 slide,slow,sod,speak,speed,
                                 start,state,steer,stop,strike,
                                 swerve,takepath,takepicture,
                                 think,ticket,transport,try,
                                 turn,want,with,yell,yield])]]]],
[_64520,[[first,glarf],[rest,_64514]]],
[_64514,t],
[_65012,[[dtrs,[[headdtr,_65026],
                [sprdtrs,[[first,_65038],[rest,end]]]]],
         [maj,_65050],
         [pecParseTree,[[daughters,[[x1,_65038],[x2,_65026]]],
                        [rule_number,rn_5]]],
         [phon,[[eol,_64520],[list,_65098]]],
         [qstore,[[eol,_64502],[list,_65116]]],
         [synsem,_64478]]],
[_65050,n],
[_65026,[[dtrs,[[headdtr,_65146]]],
         [maj,_65050],
         [pecParseTree,[[daughters,[[x1,_65146]]],
                        [rule_number,rn_6]]],
         [phon,_65182],
         [qstore,_65188],
         [synsem,_65194]]],
[_65188,[[eol,_64502],[list,_64502]]],
[_65182,[[eol,_64520],[list,_65230]]],
[_65146,[[maj,_65050],
         [pecParseTree,[[daughters,[[pecPhon,driver]]],
                        [rule_number,rn_290]]],
         [phon,_65182],
         [qstore,_65188],
         [synsem,[[local,[[category,[[bar,zero],
                                      [comps,_65308],
                                      [head,_65314],
                                      [marking,_65320],
                                      [spr,_65326],
                                      [subj,_65332]]],
                          [conj,nil],
                          [content,_65344],
                          [context,_65350]]]]]],
[_65332,end],
[_65314,[[case,nom],[form,norm],[maj,_65050]]],
[_65350,[[eol,_64460],[list,_64460]]],
[_65326,[[first,[[synsem,_65416]]],[rest,end]]],
[_65308,end],
[_65194,[[local,[[category,[[bar,one],
                            [comps,_65308],
                            [head,_65314],
                            [marking,_65320],
                            [spr,_65326],
                            [subj,_65332]]],
                 [conj,nil],
                 [content,_65344],
                 [context,_65350]]]]],
[_65344,[[index,_65512],
         [restr,[[first,[[instance,_65512],[relation,driver]]],
                 [rest,end]]]]],
[_65512,[[gend,neuter],[num,singular],[per,third]]],
[_65416,[[local,[[category,[[bar,nil],
                            [head,[[maj,_65602],
                                   [spec,[[synsem,_65194]]]]]]],
                 [conj,nil],
                 [content,_65626],
                 [context,[[eol,_64460],[list,_64460]]]]]]],
[_65602,det],
[_65320,unmarked],
[_65230,[[first,driver],[rest,_64520]]],
[_65626,[[det,definite],[restind,_65344]]],
[_65038,[[maj,_65602],
         [pecParseTree,[[daughters,[[pecPhon,the]]],
                        [rule_number,rn_219]]],
         [phon,[[eol,_65230],[list,_65098]]],
         [qstore,[[eol,_64502],[list,_65116]]],
         [synsem,_65416]]],
[_64478,[[local,[[category,[[bar,two],
                            [comps,end],
                            [head,_65314],
                            [marking,_65320],
                            [spr,end],
                            [subj,_65332]]],
                 [conj,nil],
                                [content,_65344],
                                [context,[[eol,_64460],[list,_64460]]]]]]],
               [_64460,t],
               [_65116,[[first,_65626],[rest,_64502]]],
               [_64502,t],
               [_65098,[[first,the],[rest,_65230]]],
               [_64454,t],
               [_64496,t],
               [_64284,[[dtrs,[[headdtr,_64320],
                               [subjdtrs,[[first,_65012],[rest,end]]]]],
                        [maj,_64310],
                        [pecParseTree,[[daughters,[[x1,_65012],[x2,_64320]]],
                                       [rule_number,rn_0]]],
                        [phon,[[eol,_64514],[list,_65098]]],
                        [qstore,[[eol,_64496],[list,_65116]]],
                        [synsem,[[local,[[category,[[bar,two],
                                                   [comps,end],
                                                   [head,_64296],
                                                   [spr,end],
                                                   [subj,end]]],
                                        [conj,nil],
                                        [content,_64436],
                                        [context,[[eol,_64454],
                                                  [list,_64460]]]]]]]]]]])]]]
```

E.1.7 Transitive Verb

```
Input:  parsing"glarf numi" as a VP
Result:  glarf is a transitive verb
```

```
[[lex_rule([s(0),
            [glarf|_8118],
            [[[maj,v],
              [pecParseTree,[[daughters,[[word,glarf]]],
                            [rule_number,rn_1001]]],
              [phon,[[eol,_8119],[list,[[first,glarf],[rest,_8119]]]]],
              [synsem,[[local,[[category,[[bar,zero]]],
                                         [content,
                                          [[psoa,[[relation,_8120]]]]]]]]]],
             [[_8119,t],
              [_8120,t]]]]),
  [lex_rule([s(0),
             [glarf|_8118],
[[[maj,v],
  [pecParseTree,[[daughters,[[word,glarf]]],
                 [rule_number,rn_1001]]],
  [phon,[[eol,_69116],[list,_69686]]],
  [qstore,[[eol,_69092],[list,_69704]]],
  [synsem,[[local,[[category,[[bar,zero],
                              [comps,[[first,[[synsem,_69074]]],
                                      [rest,end]]],
                              [head,_69758],
                              [subj,_69764]]],
                   [conj,nil],
                   [content,_69776],
                   [context,[[eol,_69284],[list,_69794]]]]]]]],
 [[_69006,v],
  [_69012,[[dtrs,[[headdtr,_69026]]],
           [maj,_69032],
           [pecParseTree,[[daughters,[[x1,_69026]]],
                          [rule_number,rn_1]]],
           [phon,_69062],
           [qstore,_69068],
           [synsem,_69074]]],
  [_69032,n],
  [_69068,[[eol,_69092],[list,_69092]]],
  [_69062,[[eol,_69110],[list,_69116]]],
  [_69026,[[dtrs,[[headdtr,_69134]]],
           [maj,_69032],
           [pecParseTree,[[daughters,[[x1,_69134]]],
                          [rule_number,rn_6]]],
           [phon,_69062],
           [qstore,_69068],
           [synsem,[[local,[[category,[[bar,one],
                                       [comps,_69206],
                                       [head,_69212],
                                       [marking,_69218],
                                       [spr,_69224],
                                       [subj,_69230]]],
                            [conj,nil],
                            [content,_69242],
                            [context,_69248]]]]]]],
  [_69248,[[backgr,[[first,[[name,num1]]]]],
           [eol,_69278],
           [list,_69284]]],
  [_69212,[[case,acc],[form,norm],[maj,_69032]]],
  [_69224,end],
[_69206,end],
[_69134,[[maj,_69032],
         [pecParseTree,[[daughters,[[pecPhon,numi]]],
                        [rule_number,rn_238]]],
         [phon,_69062],
         [qstore,_69068],
         [synsem,[[local,[[category,[[bar,zero],
                                     [comps,_69206],
                                     [head,_69212],
                                     [marking,_69218],
                                     [spr,_69224],
                                     [subj,_69230]]],
                          [conj,nil],
                          [content,_69242],
                          [context,_69248]]]]]]],
[_69218,unmarked],
[_69242,[[index,_69458]]],
[_69458,[[gend,neuter],[num,singular],[per,third]]],
[_69230,t],
[_69074,[[local,[[category,[[bar,two],
                            [comps,end],
                            [head,_69212],
                            [spr,end],
                            [subj,_69230]]],
                 [conj,nil],
                 [content,_69242],
                 [context,_69248]]]]],
[_69284,[[first,[[backgr,[[first,[[bearer,_69458],
                                  [relation,naming]]]]]]],
                          [rest,_69278]]],
[_69278,t],
[_69092,t],
[_69116,[[first,numi],[rest,_69110]]],
[_69110,t],
[_69636,[[maj,v],
         [pecParseTree,[[daughters,[[word,glarf]]],
                        [rule_number,rn_1001]]],
         [phon,[[eol,_69116],[list,_69686]]],
         [qstore,[[eol,_69092],[list,_69704]]],
         [synsem,[[local,[[category,[[bar,zero],
                                     [comps,[[first,
                                              [[synsem,_69074]]],
                                             [rest,end]]],
                                     [head,_69758],
                                     [subj,_69764]]],
                          [conj,nil],
                          [content,_69776],
                          [context,[[eol,_69284],
                                    [list,_69794]]]]]]]]],
[_69776,[[psoa,[[relation,disj([admit,against,approach,arrive,
                                at,avoid,back,bear,beat,become,
                                behind,bend,brake,by,carry,
                                catchfire,collide,come,
                                complete,compress,continue,
                                damage,dest,destroy,drive,
                                enter,exists,fail,flip,follow,
                                from,give,go,halt,hasproperty,
                                have,head,hit,impact,in,
                                injure,inspect,into,kill,loc,
                                lose,merge,of,over,own,park,
                                partitive,proceed,pull,ram,
                                reach,reshape,run,runover,
                                rust,see,show,skid,slide,
                                slow,spin,start,state,steer,
                                stop,strike,sustain,swerve,
                                ticket,transport,try,turn,
                                vehicle,want,with,yell,
                                yield])]]]]],
[_69686,[[first,glarf],[rest,_69116]]],
[_69758,[[maj,_69006]]],
[_69764,t],
[_69794,t],
[_69704,t],
[_68994,[[dtrs,[[compdtrs,[[first,_69012],[rest,end]]],
                [headdtr,_69636]]],
         [maj,_69006],
         [pecParseTree,[[daughters,[[x1,_69636],[x2,_69012]]],
                        [rule_number,rn_16]]],
         [phon,[[eol,_69110],[list,_69686]]],
         [qstore,[[eol,_69092],[list,_69704]]],
         [synsem,[[local,[[category,[[bar,one],
                                     [comps,end],
                                     [head,_69758],
                                     [subj,_69764]]],
                          [conj,nil],
                          [content,_69776],
                          [context,[[eol,_69278],
                                    [list,_69794]]]]]]]]]]]])]]]
```

E.2 Converging on a Lexical Entry through Multiple Inputs

This section demonstrates the use of information from multiple inputs to converge on a lexical entry for a new word. The first two subsections give the lexical version hyperspace which results from each of two pieces of input, and the third shows their merger into a single, more restricted definition. Subsequent subsections show the resulting of merging this definition with that obtained from further pieces of input.

E.2.1 The driver glarf left

```
[[lex_rule([s(0),
            [glarf|_15334],
            [[[maj,adj],
              [pecParseTree,[[daughters,[[word,glarf]]],[rule_number,rn_1001]]],
              [phon,[[eol,_15335],[list,[[first,glarf],[rest,_15335]]]]],
              [synsem,[[local,[[category,[[bar,zero],
                                          [comps,end],
                                          [head,
                                           [[maj,adj],
                                            [mod,
                                             [[synsem,
                                               [[local,
                                                 [[content,
                                                   [[index,_15336],
                                                    [restr,
                                                     [[first,_15337],
                                                      [rest,_15338]]]]]]]]]]]]],
                                          [subj,t]]],
                               [content,[[index,_15336],
                                         [restr,[[first,_15337],
                                                 [rest,[[first,
                                                         [[arg,_15339],
                                                          [relation,_15340]]],
                                                        [rest,_15338]]]]]]]]]]],
             [[_15336,_15339],
              [_15335,t],
              [_15337,t],
              [_15339,t],
              [_15338,t],
              [_15340,t]]]]),
  [lex_rule([s(0),
[glarf|_15334],
[[[maj,adj],
  [pecParseTree,[[daughters,[[word,glarf]]],[rule_number,rn_1001]]],
  [phon,_68000],
  [qstore,_68006],
  [synsem,[[local,[[category,[[bar,zero],
                              [comps,_68246],
                              [head,_68042],
                              [marking,_68258],
                              [spr,_68264],
                              [subj,_68054]]],
                   [conj,nil],
                   [content,_67408],
                   [context,_68072]]]]]],
 [[_66890,[[form,finite],[maj,_66904]]],
  [_66904,v],
  [_66914,[[dtrs,[[headdtr,_66928]]],
            [maj,_66904],
            [pecParseTree,[[daughters,[[x1,_66928]]],
                           [rule_number,rn_2]]],
            [phon,_66964],
            [qstore,_66970],
            [synsem,[[local,[[category,[[bar,two],
                                        [comps,end],
                                        [head,_66890],
                                        [spr,end],
                                        [subj,_67018]]],
                             [conj,nil],
                             [content,_67030],
                             [context,_67036]]]]]]],
  [_67036,[[eol,_67048],[list,_67048]]],
  [_67018,[[first,[[synsem,_67072]]],[rest,end]]],
  [_66970,[[eol,_67090],[list,_67090]]],
  [_66964,[[eol,_67108],[list,_67114]]],
  [_66928,[[dtrs,[[headdtr,_67132]]],
            [maj,_66904],
            [pecParseTree,[[daughters,[[x1,_67132]]],
                           [rule_number,rn_7]]],
            [phon,_66964],
            [qstore,_66970],
            [synsem,[[local,[[category,[[bar,one],
                                        [comps,_67204],
                                        [head,_66890],
                                        [marking,_67216],
                                     [spr,_67222],
                                     [subj,_67018]]],
                          [conj,nil],
                          [content,_67030],
                          [context,_67036]]]]]],
[_67222,end],
[_67204,end],
[_67132,[[maj,_66904],
         [pecParseTree,[[daughters,[[pecPhon,left]]],
                        [rule_number,rn_538]]],
         [phon,_66964],
         [qstore,_66970],
         [synsem,[[local,[[category,[[bar,zero],
                                     [comps,_67204],
                                     [head,_66890],
                                     [marking,_67216],
                                     [spr,_67222],
                                     [subj,_67018]]],
                          [conj,nil],
                          [content,_67030],
                          [context,_67036]]]]]],
[_67216,t],
[_67048,t],
[_67030,[[psoa,[[leaver,_67408],[relation,leave]]],
         [restr,end],
         [tense,past]]],
[_67072,[[local,[[category,[[bar,two],
                            [comps,end],
                            [head,_67462],
                            [marking,_67468],
                            [spr,end],
                            [subj,_67460]]],
                 [conj,nil],
                 [content,_67408],
                 [context,[[eol,_67048],[list,_67510]]]]]]],
[_67408,[[index,_67522],
         [restr,[[first,_67534],
                 [rest,
                  [[first,
                    [[arg,_67522],
                     [relation,disj([accordingto,belligerent,
                                     dead,disinterested,
                                     eastbound,further,here,
                                     just,little,loc,northbound,
                                     of,on,partitive,sideways,
                                     southbound,speed,still,
                                     supposedly,unable,
                                     westbound,with])]]],
                        [rest,_67610]]]]]]],
[_67480,end],
[_67462,[[case,nom],[form,norm],[maj,_67640]]],
[_67640,n],
[_67090,t],
[_67114,[[first,left],[rest,_67108]]],
[_67108,t],
[_67610,end],
[_67534,[[instance,_67522],[relation,driver]]],
[_67522,[[gend,neuter],[num,singular],[per,third]]],
[_67468,unmarked],
[_67734,[[dtrs,[[headdtr,_67748],
                [sprdtrs,[[first,_67760],[rest,end]]]]],
         [maj,_67640],
         [pecParseTree,[[daughters,[[x1,_67760],[x2,_67748]]],
                        [rule_number,rn_5]]],
         [phon,[[eol,_67114],[list,_67820]]],
         [qstore,[[eol,_67090],[list,_67838]]],
         [synsem,_67072]]],
[_67748,[[dtrs,[[adjdtr,_67862],[headdtr,_67868]]],
         [maj,_67640],
         [pecParseTree,[[daughters,[[x1,_67868],[x2,_67862]]],
                        [rule_number,rn_20]]],
         [phon,[[eol,_67114],[list,_67922]]],
         [qstore,[[eol,_67090],[list,_67940]]],
         [synsem,_67946]]],
[_67862,[[dtrs,[[headdtr,_67964]]],
         [maj,_67970],
         [pecParseTree,[[daughters,[[x1,_67964]]],
                        [rule_number,rn_3]]],
         [phon,_68000],
         [qstore,_68006],
         [synsem,[[local,[[category,[[bar,two],
                                     [comps,end],
                                     [head,_68042],
                                     [spr,end],
                                     [subj,_68054]]],
                          [conj,nil],
                          [content,_67408],
                          [context,_68072]]]]]]],
[_68072,[[eol,_67048],[list,_67510]]],
[_68042,[[maj,_67970],[mod,[[synsem,_68114]]]]],
[_68006,[[eol,_67090],[list,_67940]]],
[_68000,[[eol,_67114],[list,_68150]]],
[_67970,adj],
[_67964,[[dtrs,[[headdtr,_68174]]],
         [maj,_67970],
         [pecParseTree,[[daughters,[[x1,_68174]]],
                        [rule_number,rn_8]]],
         [phon,_68000],
         [qstore,_68006],
         [synsem,[[local,[[category,[[bar,one],
                                     [comps,_68246],
                                     [head,_68042],
                                     [marking,_68258],
                                     [spr,_68264],
                                     [subj,_68054]]],
                          [conj,nil],
                          [content,_67408],
                          [context,_68072]]]]]]],
[_68264,end],
[_68246,end],
[_68174,[[maj,adj],
         [pecParseTree,[[daughters,[[word,glarf]]],
                        [rule_number,rn_1001]]],
         [phon,_68000],
         [qstore,_68006],
         [synsem,[[local,[[category,[[bar,zero],
                                     [comps,_68246],
                                     [head,_68042],
                                     [marking,_68258],
                                     [spr,_68264],
                                     [subj,_68054]]],
                          [conj,nil],
                          [content,_67408],
                          [context,_68072]]]]]]],
[_68258,t],
[_68054,t],
[_68114,[[local,[[category,[[bar,_68456],
                            [comps,_68462],
                            [head,_67462],
                            [marking,_67468],
                            [spr,_68480],
                            [subj,_67480]]],
                 [conj,_68492],
                 [content,_68498],
                 [context,_68504]]]]],
[_68492,nil],
[_68480,[[first,[[synsem,_68528]]],[rest,end]]],
[_68462,end],
[_68456,one],
[_68498,[[index,_67522],[restr,[[first,_67534],[rest,_67610]]]]],
[_68150,[[first,glarf],[rest,_67114]]],
[_68504,[[eol,_67510],[list,_67510]]],
[_67868,[[dtrs,[[headdtr,_68630]]],
         [maj,_67640],
         [pecParseTree,[[daughters,[[x1,_68630]]],
                        [rule_number,rn_6]]],
         [phon,_68666],
         [qstore,_68672],
         [synsem,_68114]]],
[_68672,[[eol,_67940],[list,_67940]]],
[_68666,[[eol,_68150],[list,_67922]]],
[_68630,[[maj,_67640],
         [pecParseTree,[[daughters,[[pecPhon,driver]]],
                        [rule_number,rn_290]]],
         [phon,_68666],
         [qstore,_68672],
         [synsem,[[local,[[category,[[bar,zero],
                                     [comps,_68462],
                                     [head,_67462],
                                     [marking,_67468],
                                     [spr,_68480],
                                     [subj,_67480]]],
                          [conj,nil],
                          [content,_68498],
                          [context,_68504]]]]]]],
[_67946,[[local,[[category,[[bar,_68456],
                            [comps,_68462],
                            [head,_67462],
                            [marking,_67468],
                            [spr,_68480],
                            [subj,_67480]]],
                 [conj,_68492],
                 [content,_67408],
                 [context,[[eol,_67048],[list,_67510]]]]]]],
[_68528,[[local,[[category,[[bar,nil],
                            [head,[[maj,_68954],
                                                  [spec,[[synsem,_67946]]]]]],
                                [conj,nil],
                                [content,_68978],
                                [context,[[eol,_67510],[list,_67510]]]]]],
              [_68954,det],
              [_67940,t],
              [_67922,[[first,driver],[rest,_68150]]],
              [_68978,[[det,definite],[restind,_67408]]],
              [_67760,[[maj,_68954],
                        [pecParseTree,[[daughters,[[pecPhon,the]]],
                                       [rule_number,rn_219]]],
                        [phon,[[eol,_67922],[list,_67820]]],
                        [qstore,[[eol,_67940],[list,_67838]]],
                        [synsem,_68528]]],
              [_67510,t],
              [_67838,[[first,_68978],[rest,_67940]]],
              [_67820,[[first,the],[rest,_67922]]],
              [_66878,[[dtrs,[[headdtr,_66914],
                               [subjdtrs,[[first,_67734],[rest,end]]]]],
                        [maj,_66904],
                        [pecParseTree,[[daughters,[[x1,_67734],[x2,_66914]]],
                                       [rule_number,rn_0]]],
                        [phon,[[eol,_67108],[list,_67820]]],
                        [qstore,[[eol,_67090],[list,_67838]]],
                        [synsem,[[local,[[category,[[bar,two],
                                                    [comps,end],
                                                    [head,_66890],
                                                    [spr,end],
                                                    [subj,end]]],
                                         [conj,nil],
                                         [content,_67030],
                                         [context,[[eol,_67048],
                                                   [list,_67510]]]]]]]]]]])]],
[lex_rule([s(0),
           [glarf|_317711],
           [[[maj,adv],
             [pecParseTree,[[daughters,[[word,glarf]]],[rule_number,rn_1001]]],
             [phon,[[eol,_317712],[list,[[first,glarf],[rest,_317712]]]]],
             [synsem,[[local,[[category,[[bar,zero],
                                          [comps,end],
                                          [head,[[maj,adv],
                                                 [mod,
                                                  [[synsem,
                                                    [[local,
                                                    [[content,
                                                      [[psoa,_317713],
                                                       [restr,_317714]]]]]]]]]],
                                        [subj,t]]],
                             [content,[[psoa,_317713],
                                       [restr,[[first,[[relation,_317715],
                                                       [soaarg,_317716]]],
                                               [rest,_317714]]]]]]]]],
           [[_317713,_317716],
            [_317712,t],
            [_317716,t],
            [_317714,t],
            [_317715,t]]]]),
[lex_rule([s(0),
           [glarf|_317711],
            [[[maj,adv],
              [pecParseTree,[[daughters,[[word,glarf]]],[rule_number,rn_1001]]],
              [phon,_366640],[qstore,_366646],
              [synsem,[[local,[[category,[[bar,zero],
                                          [comps,_366886],
                                          [head,_366682],
                                          [marking,_366898],
                                          [spr,_366904],
                                          [subj,_366694]]],
                                [conj,nil],
                                [content,_365950],
                                [context,_366712]]]]]],
             [[_365810,[[form,finite],[maj,_365824]]],
              [_365824,v],
              [_365834,[[dtrs,[[headdtr,_365848]]],
                        [maj,_365824],
                        [pecParseTree,[[daughters,[[x1,_365848]]],
                                       [rule_number,rn_2]]],
                        [phon,_365884],
                        [qstore,_365890],
                        [synsem,[[local,[[category,[[bar,two],
                                                    [comps,end],
                                                    [head,_365810],
                                                    [spr,end],
                                                    [subj,_365938]]],
                                         [conj,nil],
                                         [content,_365950],
                                         [context,_365956]]]]]]],
              [_365956,[[eol,_365968],[list,_365974]]],
[_365938,[[first,[[synsem,_365992]]],[rest,end]]],
[_365890,[[eol,_366010],[list,_366016]]],
[_365884,[[eol,_366028],[list,_366034]]],
[_365848,[[dtrs,[[adjdtr,_366052],[headdtr,_366058]]],
          [maj,_365824],
          [pecParseTree,[[daughters,[[x1,_366052],[x2,_366058]]],
                         [rule_number,rn_22]]],
          [phon,_365884],
          [qstore,_365890],
          [synsem,[[local,[[category,[[bar,_366130],
                                      [comps,_366136],
                                      [head,_365810],
                                      [marking,_366148],
                                      [spr,_366154],
                                      [subj,_365938]]],
                           [conj,_366166],
                           [content,_365950],
                           [context,_365956]]]]]]],
[_366166,nil],
[_366154,end],
[_366136,end],
[_366130,one],
[_366058,[[dtrs,[[headdtr,_366220]]],
          [maj,_365824],
          [pecParseTree,[[daughters,[[x1,_366220]]],
                         [rule_number,rn_7]]],
          [phon,_366256],
          [qstore,_366262],
          [synsem,_366268]]],
[_366262,[[eol,_366010],[list,_366010]]],
[_366256,[[eol,_366028],[list,_366304]]],
[_366220,[[maj,_365824],
          [pecParseTree,[[daughters,[[pecPhon,left]]],
                         [rule_number,rn_538]]],
          [phon,_366256],
          [qstore,_366262],
          [synsem,[[local,[[category,[[bar,zero],
                                      [comps,_366136],
                                      [head,_365810],
                                      [marking,_366148],
                                      [spr,_366154],
                                      [subj,_365938]]],
                           [conj,nil],
                           [content,_366418],
                           [context,_366424]]]]]]],
[_366424,[[eol,_365968],[list,_365968]]],
[_366418,[[psoa,_366454],[restr,_366460],[tense,past]]],
[_366268,[[local,[[category,[[bar,_366130],
                             [comps,_366136],
                             [head,_365810],
                             [marking,_366148],
                             [spr,_366154],
                             [subj,_365938]]],
                  [conj,_366166],
                  [content,_366418],
                  [context,_366424]]]]],
[_366460,end],
[_366454,[[leaver,_366556],[relation,leave]]],
[_366148,t],
[_366304,[[first,left],[rest,_366028]]],
[_366052,[[dtrs,[[headdtr,_366604]]],
          [maj,_366610],
          [pecParseTree,[[daughters,[[x1,_366604]]],
                         [rule_number,rn_4]]],
          [phon,_366640],
          [qstore,_366646],
          [synsem,[[local,[[category,[[bar,two],
                                      [comps,end],
                                      [head,_366682],
                                      [spr,end],
                                      [subj,_366694]]],
                           [conj,nil],
                           [content,_365950],
                           [context,_366712]]]]]]],
[_366712,[[eol,_365968],[list,_365974]]],
[_366682,[[maj,_366610],[mod,[[synsem,_366268]]]]],
[_366646,[[eol,_366010],[list,_366016]]],
[_366640,[[eol,_366304],[list,_366034]]],
[_366610,adv],
[_366604,[[dtrs,[[headdtr,_366814]]],
          [maj,_366610],
          [pecParseTree,[[daughters,[[x1,_366814]]],
                         [rule_number,rn_9]]],
          [phon,_366640],
          [qstore,_366646],
          [synsem,[[local,[[category,[[bar,one],
                                      [comps,_366886],
                                      [head,_366682],
                                      [marking,_366898],
                                      [spr,_366904],
                                      [subj,_366694]]],
                           [conj,nil],
                           [content,_365950],
                           [context,_366712]]]]]]],
[_366904,end],
[_366886,end],
[_366814,[[maj,adv],
          [pecParseTree,[[daughters,[[word,glarf]]],
                         [rule_number,rn_1001]]],
          [phon,_366640],
          [qstore,_366646],
          [synsem,[[local,[[category,[[bar,zero],
                                      [comps,_366886],
                                      [head,_366682],
                                      [marking,_366898],
                                      [spr,_366904],
                                      [subj,_366694]]],
                           [conj,nil],
                           [content,_365950],
                           [context,_366712]]]]]]],
[_366898,t],
[_366694,t],
[_365974,t],
[_365968,t],
[_365950,[[psoa,_366454],
          [restr,
           [[first,
             [[relation,
               disj([accordingto,across,after,against,along,
                     also,as,asfaras,at,before,behind,dest,dir,
                     down,for,from,further,here,in,just,little,
                     loc,not,of,on,over,past,path_change,
                     sideways,simultaneouswith,still,suddenly,
                     supposedly,then,through,toward,wednesday,
                     when,while,with,within,without])],
                          [soaarg,_366454]]],
                  [rest,_366460]]]]],
[_365992,[[local,[[category,[[bar,two],
                             [comps,end],
                             [head,_367248],
                             [marking,_367254],
                             [spr,end],
                              [subj,_367266]]],
                  [conj,nil],
                  [content,_366556],
                  [context,[[eol,_365974],[list,_365974]]]]]],
[_366556,[[index,_367308],
          [restr,[[first,[[instance,_367308],[relation,driver]]],
                  [rest,end]]]]],
[_367266,end],
[_367248,[[case,nom],[form,norm],[maj,_367368]]],
[_367368,n],
[_366016,t],
[_366010,t],
[_366034,[[first,glarf],[rest,_366304]]],
[_366028,t],
[_367308,[[gend,neuter],[num,singular],[per,third]]],
[_367254,unmarked],
[_367444,[[dtrs,[[headdtr,_367458],
                 [sprdtrs,[[first,_367470],[rest,end]]]]],
          [maj,_367368],
          [pecParseTree,[[daughters,[[x1,_367470],[x2,_367458]]],
                         [rule_number,rn_5]]],
          [phon,[[eol,_366034],[list,_367530]]],
          [qstore,[[eol,_366016],[list,_367548]]],
          [synsem,_365992]]],
[_367458,[[dtrs,[[headdtr,_367572]]],
          [maj,_367368],
          [pecParseTree,[[daughters,[[x1,_367572]]],
                         [rule_number,rn_6]]],
          [phon,_367608],
          [qstore,_367614],
          [synsem,_367620]]],
[_367614,[[eol,_366016],[list,_366016]]],
[_367608,[[eol,_366034],[list,_367656]]],
[_367572,[[maj,_367368],
          [pecParseTree,[[daughters,[[pecPhon,driver]]],
                         [rule_number,rn_290]]],
          [phon,_367608],
          [qstore,_367614],
          [synsem,[[local,[[category,[[bar,zero],
                                      [comps,_367734],
                                      [head,_367248],
                                      [marking,_367254],
                                      [spr,_367752],
                                      [subj,_367266]]],
                           [conj,nil],
                           [content,_366556],
                           [context,_367776]]]]]],
[_367776,[[eol,_365974],[list,_365974]]],
[_367752,[[first,[[synsem,_367812]]],[rest,end]]],
[_367734,end],
[_367620,[[local,[[category,[[bar,one],
                             [comps,_367734],
                             [head,_367248],
                             [marking,_367254],
                             [spr,_367752],
                             [subj,_367266]]],
                  [conj,nil],
                  [content,_366556],
                  [context,_367776]]]]],
[_367812,[[local,[[category,[[bar,nil],
                             [head,[[maj,_367932],
                                    [spec,[[synsem,_367620]]]]]]],
                  [conj,nil],
                  [content,_367956],
                  [context,[[eol,_365974],[list,_365974]]]]]]],
[_367932,det],
[_367656,[[first,driver],[rest,_366034]]],
[_367956,[[det,definite],[restind,_366556]]],
[_367470,[[maj,_367932]
          [pecParseTree,[[daughters,[[pecPhon,the]]],
                         [rule_number,rn_219]]],
          [phon,[[eol,_367656],[list,_367530]]],
          [qstore,[[eol,_366016],[list,_367548]]],
          [synsem,_367812]]],
[_367548,[[first,_367956],[rest,_366016]]],
[_367530,[[first,the],[rest,_367656]]],
[_365798,[[dtrs,[[headdtr,_365834],
                 [subjdtrs,[[first,_367444],[rest,end]]]]],
          [maj,_365824],
          [pecParseTree,[[daughters,[[x1,_367444],[x2,_365834]]],
                         [rule_number,rn_0]]],
          [phon,[[eol,_366028],[list,_367530]]],
          [qstore,[[eol,_366010],[list,_367548]]],
          [synsem,[[local,[[category,[[bar,two],
                                      [comps,end],
                                      [head,_365810],
                                      [spr,end],
                                      [subj,end]]],
                                            [conj,nil],
                                            [content,_365950],
                                            [context,[[eol,_365968],
                                                      [list,_365974]]]]]]]]]]]])]]]
```

E.2.2 The driver glarf stopped

```
[[lex_rule([s(0),
            [glarf|_15335],
            [[[maj,adj],
              [pecParseTree,[[daughters,[[word,glarf]]],[rule_number,rn_1001]]],
              [phon,[[eol,_15336],[list,[[first,glarf],[rest,_15336]]]]],
              [synsem,[[local,[[category,[[bar,zero],
                                           [comps,end],
                                           [head,
                                            [[maj,adj],
                                             [mod,
                                              [[synsem,
                                                [[local,
                                                  [[content,
                                                    [[index,_15337],
                                                     [restr,
                                                      [[first,_15338],
                                                       [rest,_15339]]]]]]]]]]]],
                                           [subj,t]]],
                                [content,[[index,_15337],
                                          [restr,[[first,_15338],
                                                  [rest,[[first,
                                                          [[arg,_15340],
                                                           [relation,_15341]]],
                                                         [rest,_15339]]]]]]]]]]],
             [[_15337,_15340],
              [_15336,t],
              [_15338,t],
              [_15340,t],
              [_15339,t],
              [_15341,t]]]]),
  [lex_rule([s(0),
             [glarf|_15335],
             [[[maj,adj],
               [pecParseTree,[[daughters,[[word,glarf]]],[rule_number,rn_1001]]],
               [phon,_63982],
               [qstore,_63988],
 [synsem,[[local,[[category,[[bar,zero],
                             [comps,_64228],
                             [head,_64024],
                             [marking,_64240],
                             [spr,_64246],
                             [subj,_64036]]],
                  [conj,nil],
                  [content,_63396],
                  [context,_64054]]]]]],
[[_62872,[[form,finite],[maj,_62886]]],
 [_62886,v],
 [_62896,[[dtrs,[[headdtr,_62910]]],
          [maj,_62886],
          [pecParseTree,[[daughters,[[x1,_62910]]],
                         [rule_number,rn_2]]],
          [phon,_62946],
          [qstore,_62952],
          [synsem,[[local,[[category,[[bar,two],
                                      [comps,end],
                                      [head,_62872],
                                      [spr,end],
                                      [subj,_63000]]],
                           [conj,nil],
                           [content,_63012],
                           [context,_63018]]]]]],
 [_63018,[[eol,_63030],[list,_63030]]],
 [_63000,[[first,[[synsem,_63054]]],[rest,end]]],
 [_62952,[[eol,_63072],[list,_63072]]],
 [_62946,[[eol,_63090],[list,_63096]]],
 [_62910,[[dtrs,[[headdtr,_63114]]],
          [maj,_62886],
          [pecParseTree,[[daughters,[[x1,_63114]]],
                         [rule_number,rn_7]]],
          [phon,_62946],
          [qstore,_62952],
          [synsem,[[local,[[category,[[bar,one],
                                      [comps,_63186],
                                      [head,_62872],
                                      [marking,_63198],
                                      [spr,_63204],
                                      [subj,_63000]]],
                           [conj,nil],
                           [content,_63012],
                           [context,_63018]]]]]],
[_63204,end],
[_63186,end],
[_63114,[[maj,_62886],
           [pecParseTree,[[daughters,[[pecPhon,stopped]]],
                            [rule_number,rn_599]]],
           [phon,_62946],
           [qstore,_62952],
           [synsem,[[local,[[category,[[bar,zero],
                                           [comps,_63186],
                                           [head,_62872],
                                           [marking,_63198],
                                           [spr,_63204],
                                           [subj,_63000]]],
                              [conj,nil],
                              [content,_63012],
                              [context,_63018]]]]]]],
[_63198,t],
[_63030,t],
[_63012,[[psoa,[[relation,stop],[stopper,_63396]]],
           [restr,end],
           [tense,past]]],
[_63054,[[local,[[category,[[bar,two],
                                [comps,end],
                                [head,_63444],
                                [marking,_63450],
                                [spr,end],
                                [subj,_63462]]],
                    [conj,nil],
                    [content,_63396],
                    [context,[[eol,_63030],[list,_63492]]]]]]],
[_63396,[[index,_63504],
           [restr,[[first,_63516],
                    [rest,[[first,
                             [[arg,_63504],
                              [relation,disj([accordingto,belligerent,
                                                dead,disinterested,
                                                eastbound,further,
                                                here,just,little,loc,
                                                northbound,of,on,
                                                partitive,sideways,
                                                southbound,speed,
                                                still,supposedly,
                                                unable,westbound,
                                                with])]]],
                        [rest,_63592]]]]]],
[_63462,end],
[_63444,[[case,nom],[form,norm],[maj,_63622]]],
[_63622,n],
[_63072,t],
[_63096,[[first,stopped],[rest,_63090]]],
[_63090,t],
[_63592,end],
[_63516,[[instance,_63504],[relation,driver]]],
[_63504,[[gend,neuter],[num,singular],[per,third]]],
[_63450,unmarked],
[_63716,[[dtrs,[[headdtr,_63730],
                [sprdtrs,[[first,_63742],[rest,end]]]]],
         [maj,_63622],
         [pecParseTree,[[daughters,[[x1,_63742],[x2,_63730]]],
                        [rule_number,rn_5]]],
         [phon,[[eol,_63096],[list,_63802]]],
         [qstore,[[eol,_63072],[list,_63820]]],
         [synsem,_63054]]],
[_63730,[[dtrs,[[adjdtr,_63844],[headdtr,_63850]]],
         [maj,_63622],
         [pecParseTree,[[daughters,[[x1,_63850],[x2,_63844]]],
                        [rule_number,rn_20]]],
         [phon,[[eol,_63096],[list,_63904]]],
         [qstore,[[eol,_63072],[list,_63922]]],
         [synsem,_63928]]],
[_63844,[[dtrs,[[headdtr,_63946]]],
         [maj,_63952],
         [pecParseTree,[[daughters,[[x1,_63946]]],
                        [rule_number,rn_3]]],
         [phon,_63982],
         [qstore,_63988],
         [synsem,[[local,[[category,[[bar,two],
                                    [comps,end],
                                    [head,_64024],
                                    [spr,end],
                                    [subj,_64036]]],
                          [conj,nil],
                          [content,_63396],
                          [context,_64054]]]]]]],
[_64054,[[eol,_63030],[list,_63492]]],
[_64024,[[maj,_63952],[mod,[[synsem,_64096]]]]],
[_63988,[[eol,_63072],[list,_63922]]],
[_63982,[[eol,_63096],[list,_64132]]],
[_63952,adj],
[_63946,[[dtrs,[[headdtr,_64156]]],
         [maj,_63952],
         [pecParseTree,[[daughters,[[x1,_64156]]],
                        [rule_number,rn_8]]],
         [phon,_63982],
         [qstore,_63988],
         [synsem,[[local,[[category,[[bar,one],
                                     [comps,_64228],
                                     [head,_64024],
                                     [marking,_64240],
                                     [spr,_64246],
                                     [subj,_64036]]],
                          [conj,nil],
                          [content,_63396],
                          [context,_64054]]]]]]],
[_64246,end],
[_64228,end],
[_64156,[[maj,adj],
         [pecParseTree,[[daughters,[[word,glarf]]],
                        [rule_number,rn_1001]]],
         [phon,_63982],
         [qstore,_63988],
         [synsem,[[local,[[category,[[bar,zero],
                                     [comps,_64228],
                                     [head,_64024],
                                     [marking,_64240],
                                     [spr,_64246],
                                     [subj,_64036]]],
                          [conj,nil],
                          [content,_63396],
                          [context,_64054]]]]]]],
[_64240,t],
[_64036,t],
[_64096,[[local,[[category,[[bar,_64438],
                            [comps,_64444],
                            [head,_63444],
                            [marking,_63450],
                            [spr,_64462],
                            [subj,_63462]]],
                 [conj,_64474],
                 [content,_64480],
                 [context,_64486]]]]],
[_64474,nil],
[_64462,[[first,[[synsem,_64510]]],[rest,end]]],
[_64444,end],
[_64438,one],
[_64480,[[index,_63504],[restr,[[first,_63516],[rest,_63592]]]]],
[_64132,[[first,glarf],[rest,_63096]]],
[_64486,[[eol,_63492],[list,_63492]]],
[_63850,[[dtrs,[[headdtr,_64612]]],
         [maj,_63622],
         [pecParseTree,[[daughters,[[x1,_64612]]],
                        [rule_number,rn_6]]],
         [phon,_64648],
         [qstore,_64654],
         [synsem,_64096]]],
[_64654,[[eol,_63922],[list,_63922]]],
[_64648,[[eol,_64132],[list,_63904]]],
[_64612,[[maj,_63622],
         [pecParseTree,[[daughters,[[pecPhon,driver]]],
                        [rule_number,rn_290]]],
         [phon,_64648],
         [qstore,_64654],
         [synsem,[[local,[[category,[[bar,zero],
                                     [comps,_64444],
                                     [head,_63444],
                                     [marking,_63450],
                                     [spr,_64462],
                                     [subj,_63462]]],
                          [conj,nil],
                          [content,_64480],
                          [context,_64486]]]]]]],
[_63928,[[local,[[category,[[bar,_64438],
                            [comps,_64444],
                            [head,_63444],
                            [marking,_63450],
                            [spr,_64462],
                            [subj,_63462]]],
                 [conj,_64474],
                 [content,_63396],
                 [context,[[eol,_63030],[list,_63492]]]]]]],
[_64510,[[local,[[category,[[bar,nil],
                            [head,[[maj,_64936],
                                   [spec,[[synsem,_63928]]]]]]],
                 [conj,nil],
                 [content,_64960],
                 [context,[[eol,_63492],[list,_63492]]]]]]],
              [_64936,det],
              [_63922,t],
              [_63904,[[first,driver],[rest,_64132]]],
              [_64960,[[det,definite],[restind,_63396]]],
              [_63742,[[maj,_64936],
                       [pecParseTree,[[daughters,[[pecPhon,the]]],
                                      [rule_number,rn_219]]],
                       [phon,[[eol,_63904],[list,_63802]]],
                       [qstore,[[eol,_63922],[list,_63820]]],
                       [synsem,_64510]]],
              [_63492,t],
              [_63820,[[first,_64960],[rest,_63922]]],
              [_63802,[[first,the],[rest,_63904]]],
              [_62860,[[dtrs,[[headdtr,_62896],
                              [subjdtrs,[[first,_63716],[rest,end]]]]],
                       [maj,_62886],
                       [pecParseTree,[[daughters,[[x1,_63716],[x2,_62896]]],
                                      [rule_number,rn_0]]],
                       [phon,[[eol,_63090],[list,_63802]]],
                       [qstore,[[eol,_63072],[list,_63820]]],
                       [synsem,[[local,[[category,[[bar,two],
                                                   [comps,end],
                                                   [head,_62872],
                                                   [spr,end],
                                                   [subj,end]]],
                                        [conj,nil],
                                        [content,_63012],
                                        [context,[[eol,_63030],
                                                  [list,_63492]]]]]]]]]]])]],
[lex_rule([s(0),
           [glarf|_209272],
           [[[maj,adv],
             [pecParseTree,[[daughters,[[word,glarf]]],[rule_number,rn_1001]]],
             [phon,[[eol,_209273],[list,[[first,glarf],[rest,_209273]]]]],
             [synsem,[[local,[[category,[[bar,zero],
                                         [comps,end],
                                         [head,[[maj,adv],
                                                [mod,
                                                 [[synsem,
                                                   [[local,
                                                     [[content,
                                                       [[psoa,_209274],
                                                        [restr,_209275]]]]]]]]]]],
                                         [subj,t]]],
                              [content,[[psoa,_209274],
                                        [restr,[[first,[[relation,_209276],
                                                         [soaarg,_209277]]],
                                                 [rest,_209275]]]]]]]]],
           [[_209274,_209277],
            [_209273,t],
            [_209277,t],
            [_209275,t],
            [_209276,t]]]]),
[lex_rule([s(0),
           [glarf|_209272],
           [[[maj,adv],
             [pecParseTree,[[daughters,[[word,glarf]]],[rule_number,rn_1001]]],
             [phon,_254205],
             [qstore,_254211],
             [synsem,[[local,[[category,[[bar,zero],
                                         [comps,_254451],
                                         [head,_254247],
                                         [marking,_254463],
                                         [spr,_254469],
                                         [subj,_254259]]],
                              [conj,nil],
                              [content,_253515],
                              [context,_254277]]]]]],
            [[_253375,[[form,finite],[maj,_253389]]],
             [_253389,v],
             [_253399,[[dtrs,[[headdtr,_253413]]],
                       [maj,_253389],
                       [pecParseTree,[[daughters,[[x1,_253413]]],
                                      [rule_number,rn_2]]],
                       [phon,_253449],
                       [qstore,_253455],
                       [synsem,[[local,[[category,[[bar,two],
                                                   [comps,end],
                                                   [head,_253375],
                                                   [spr,end],
                                                   [subj,_253503]]],
                                        [conj,nil],
                                        [content,_253515],
                                        [context,_253521]]]]]]],
             [_253521,[[eol,_253533],[list,_253539]]],
             [_253503,[[first,[[synsem,_253557]]],[rest,end]]],
             [_253455,[[eol,_253575],[list,_253581]]],
             [_253449,[[eol,_253593],[list,_253599]]],
[_253413,[[dtrs,[[adjdtr,_253617],[headdtr,_253623]]],
          [maj,_253389],
          [pecParseTree,[[daughters,[[x1,_253617],[x2,_253623]]],
                         [rule_number,rn_22]]],
          [phon,_253449],
          [qstore,_253455],
          [synsem,[[local,[[category,[[bar,_253695],
                                      [comps,_253701],
                                      [head,_253375],
                                      [marking,_253713],
                                      [spr,_253719],
                                      [subj,_253503]]],
                           [conj,_253731],
                           [content,_253515],
                           [context,_253521]]]]]]],
[_253731,nil],
[_253719,end],
[_253701,end],
[_253695,one],
[_253623,[[dtrs,[[headdtr,_253785]]],
          [maj,_253389],
          [pecParseTree,[[daughters,[[x1,_253785]]],
                         [rule_number,rn_7]]],
          [phon,_253821],
          [qstore,_253827],
          [synsem,_253833]]],
[_253827,[[eol,_253575],[list,_253575]]],
[_253821,[[eol,_253593],[list,_253869]]],
[_253785,[[maj,_253389],
          [pecParseTree,[[daughters,[[pecPhon,stopped]]],
                         [rule_number,rn_599]]],
          [phon,_253821],
          [qstore,_253827],
          [synsem,[[local,[[category,[[bar,zero],
                                      [comps,_253701],
                                      [head,_253375],
                                      [marking,_253713],
                                      [spr,_253719],
                                      [subj,_253503]]],
                           [conj,nil],
                           [content,_253983],
                           [context,_253989]]]]]]],
[_253989,[[eol,_253533],[list,_253533]]],
[_253983,[[psoa,_254019],[restr,_254025],[tense,past]]],
[_253833,[[local,[[category,[[bar,_253695],
                             [comps,_253701],
                             [head,_253375],
                             [marking,_253713],
                             [spr,_253719],
                             [subj,_253503]]],
                  [conj,_253731],
                  [content,_253983],
                  [context,_253989]]]]],
[_254025,end],
[_254019,[[relation,stop],[stopper,_254127]]],
[_253713,t],
[_253869,[[first,stopped],[rest,_253593]]],
[_253617,[[dtrs,[[headdtr,_254169]]],
          [maj,_254175],
          [pecParseTree,[[daughters,[[x1,_254169]]],
                         [rule_number,rn_4]]],
          [phon,_254205],
          [qstore,_254211],
          [synsem,[[local,[[category,[[bar,two],
                                      [comps,end],
                                      [head,_254247],
                                      [spr,end],
                                      [subj,_254259]]],
                           [conj,nil],
                           [content,_253515],
                           [context,_254277]]]]]]],
[_254277,[[eol,_253533],[list,_253539]]],
[_254247,[[maj,_254175],[mod,[[synsem,_253833]]]]],
[_254211,[[eol,_253575],[list,_253581]]],
[_254205,[[eol,_253869],[list,_253599]]],
[_254175,adv],
[_254169,[[dtrs,[[headdtr,_254379]]],
          [maj,_254175],
          [pecParseTree,[[daughters,[[x1,_254379]]],
                         [rule_number,rn_9]]],
          [phon,_254205],
          [qstore,_254211],
          [synsem,[[local,[[category,[[bar,one],
                                      [comps,_254451],
                                      [head,_254247],
                                      [marking,_254463],
                                      [spr,_254469],
                                      [subj,_254259]]],
                           [conj,nil],
                           [content,_253515],
                           [context,_254277]]]]]]],
[_254469,end],
[_254451,end],
[_254379,[[maj,adv],
          [pecParseTree,[[daughters,[[word,glarf]]],
                         [rule_number,rn_1001]]],
          [phon,_254205],
          [qstore,_254211],
          [synsem,[[local,[[category,[[bar,zero],
                                      [comps,_254451],
                                      [head,_254247],
                                      [marking,_254463],
                                      [spr,_254469],
                                      [subj,_254259]]],
                           [conj,nil],
                           [content,_253515],
                           [context,_254277]]]]]]],
[_254463,t],
[_254259,t],
[_253539,t],
[_253533,t],
[_253515,[[psoa,_254019],
          [restr,[[first,[[relation,disj([accordingto,across,
                                         after,against,along,
                                         also,as,asfaras,at,
                                         before,behind,complete,
                                         dest,dir,down,for,
                                         from,further,here,
                                         in,just,little,loc,
                                         not,of,on,over,past,
                                         path_change,sideways,
                                         simultaneouswith,
                                         still,suddenly,
                                         supposedly,then,
                                         through,toward,
                                         wednesday,when,while,
                                         with,within,without])],
                         [soaarg,_254019]]],
                  [rest,_254025]]]]],
[_253557,[[local,[[category,[[bar,two],
                             [comps,end],
                             [head,_254815],
                             [marking,_254821],
                             [spr,end],
                             [subj,_254833]]],
                  [conj,nil],
                  [content,_254127],
                  [context,[[eol,_253539],[list,_253539]]]]]],
[_254127,[[index,_254875],
          [restr,[[first,[[instance,_254875],[relation,driver]]],
                  [rest,end]]]]],
[_254833,end],
[_254815,[[case,nom],[form,norm],[maj,_254935]]],
[_254935,n],
[_253581,t],
[_253575,t],
[_253599,[[first,glarf],[rest,_253869]]],
[_253593,t],
[_254875,[[gend,neuter],[num,singular],[per,third]]],
[_254821,unmarked],
[_255011,[[dtrs,[[headdtr,_255025],
                 [sprdtrs,[[first,_255037],[rest,end]]]]],
          [maj,_254935],
          [pecParseTree,[[daughters,[[x1,_255037],[x2,_255025]]],
                         [rule_number,rn_5]]],
          [phon,[[eol,_253599],[list,_255097]]],
          [qstore,[[eol,_253581],[list,_255115]]],
          [synsem,_253557]]],
[_255025,[[dtrs,[[headdtr,_255139]]],
          [maj,_254935],
          [pecParseTree,[[daughters,[[x1,_255139]]],
                         [rule_number,rn_6]]],
          [phon,_255175],
          [qstore,_255181],
          [synsem,_255187]]],
[_255181,[[eol,_253581],[list,_253581]]],
[_255175,[[eol,_253599],[list,_255223]]],
[_255139,[[maj,_254935],
          [pecParseTree,[[daughters,[[pecPhon,driver]]],
                         [rule_number,rn_290]]],
          [phon,_255175],
          [qstore,_255181],
          [synsem,[[local,[[category,[[bar,zero],
                                      [comps,_255301],
                                      [head,_254815],
                                      [marking,_254821],
                                       [spr,_255319],
                                       [subj,_254833]]],
                            [conj,nil],
                            [content,_254127],
                            [context,_255343]]]]]],
[_255343,[[eol,_253539],[list,_253539]]],
[_255319,[[first,[[synsem,_255379]]],[rest,end]]],
[_255301,end],
[_255187,[[local,[[category,[[bar,one],
                             [comps,_255301],
                             [head,_254815],
                             [marking,_254821],
                             [spr,_255319],
                             [subj,_254833]]],
                  [conj,nil],
                  [content,_254127],
                  [context,_255343]]]]],
[_255379,[[local,[[category,[[bar,nil],
                             [head,[[maj,_255499],
                                    [spec,[[synsem,_255187]]]]]]],
                  [conj,nil],
                  [content,_255523],
                  [context,[[eol,_253539],[list,_253539]]]]]]],
[_255499,det],
[_255223,[[first,driver],[rest,_253599]]],
[_255523,[[det,definite],[restind,_254127]]],
[_255037,[[maj,_255499],
          [pecParseTree,[[daughters,[[pecPhon,the]]],
                         [rule_number,rn_219]]],
          [phon,[[eol,_255223],[list,_255097]]],
          [qstore,[[eol,_253581],[list,_255115]]],
          [synsem,_255379]]],
[_255115,[[first,_255523],[rest,_253581]]],
[_255097,[[first,the],[rest,_255223]]],
[_253363,[[dtrs,[[headdtr,_253399],
                 [subjdtrs,[[first,_2[illegible]1],[rest,end]]]]],
          [maj,_253389],
          [pecParseTree,[[daughters,[[x1,_255011],[x2,_253399]]],
                         [rule_number,rn_0]]],
          [phon,[[eol,_253593],[list,_255097]]],
          [qstore,[[eol,_253575],[list,_255115]]],
          [synsem,[[local,[[category,[[bar,two],
                                      [comps,end],
                                      [head,_253375],
```

BEST COPY AVAILABLE

```
                                                      [spr,end],
                                                      [subj,end]]],
                                           [conj,nil],
                                           [content,_253515],
                                           [context,[[eol,_253533],
                                                     [list,_253539]]]]]]]]]]]])]]]
```

E.2.3 Merge of previous two entries

```
[[lex_rule([s(0),
            [glarf|_15335],
            [[[maj,adj],
              [pecParseTree,[[daughters,[[word,glarf]]],[rule_number,rn_1001]]],
              [phon,[[eol,_15336],[list,[[first,glarf],[rest,_485473]]]]],
              [synsem,[[local,[[category,[[bar,zero],
                                           [comps,end],
                                           [head,
                                            [[maj,adj],
                                              [mod,
                                               [[synsem,
                                                 [[local,
                                                   [[content,
                                                     [[index,_15337],
                                                      [restr,
                                                       [[first,_15338],
                                                        [rest,_15339]]]]]]]]]]]]],
                                           [subj,t]]],
                                [content,[[index,_485683],
                                          [restr,[[first,_485720],
                                                  [rest,[[first,
                                                          [[arg,_485900],
                                                           [relation,_15341]]],
                                                         [rest,_485747]]]]]]]]]]]],
             [[_15341,_485927],
              [_15339,_485747],
              [_15338,_485720],
              [_15337,_15340],
              [_15340,_485683],
              [_15336,_485473],
              [_485927,t],
              [_485747,t],
              [_485720,t],
            [_485683,_485900],
            [_485900,t],
            [_485473,t]]]]]),
[lex_rule([s(0),
           [glarf|_15335],
           [[[maj,adj],
             [pecParseTree,[[daughters,[[word,glarf]]],[rule_number,rn_1001]]],
             [phon,_63982],
             [qstore,_63988],
             [synsem,[[local,[[category,[[bar,zero],
                                          [comps,_64228],
                                          [head,_64024],
                                          [marking,_64240],
                                          [spr,_64246],
                                          [subj,_64036]]],
                               [conj,nil],
                               [content,_63396],
                               [context,_64054]]]]]],
            [[_63982,_486269],
             [_486269,[[eol,_63096],[list,_64132]]],
             [_64132,_490670],
             [_490670,[[first,glarf],[rest,_63096]]],
             [_63096,_487345],
             [_487345,[[rest,_63090]]],
             [_63090,_487318],
             [_487318,t],
             [_63988,_486296],
             [_486296,[[eol,_63072],[list,_63922]]],
             [_63072,_487251],
             [_487251,t],
             [_63922,_489985],
             [_489985,t],
             [_64228,_486369],
             [_486369,end],
             [_64024,_486396],
             [_486396,[[maj,_63952],[mod,[[synsem,_64096]]]]],
             [_63952,_490103],
             [_490103,adj],
             [_64096,_490538],
             [_490538,[[local,[[category,[[bar,_64438],
                                           [comps,_64444],
                                           [head,_63444],
                                           [marking,_63450],
                                           [spr,_64462],
                              [subj,_63462]]],
                   [conj,_64474],
                   [content,_64480],
                   [context,_64486]]]]],
[_64462,_491704],
[_491704,[[first,[[synsem,_64510]]],[rest,end]]],
[_64510,_491891],
[_491891,[[local,[[category,[[bar,nil],
                              [head,[[maj,_64936],
                                     [spec,[[synsem,_63928]]]]]]],
                   [conj,nil],
                   [content,_64960],
                   [context,[[eol,_63492],[list,_63492]]]]]]],
[_64936,_493235],
[_493235,det],
[_63928,_490016],
[_490016,[[local,[[category,[[bar,_64438],
                              [comps,_64444],
                              [head,_63444],
                              [marking,_63450],
                              [spr,_64462],
                              [subj,_63462]]],
                   [conj,_64474],
                   [content,_63396],
                   [context,[[eol,_63030],[list,_63492]]]]]]],
[_64438,_491608],
[_491608,one],
[_64444,_491635],
[_491635,end],
[_63444,_488419],
[_488419,[[case,nom],[form,norm],[maj,_63622]]],
[_63622,_489012],
[_489012,n],
[_63450,_488446],
[_488446,unmarked],
[_64960,_493321],
[_493321,[[det,definite],[restind,_63396]]],
[_63462,_488489],
[_488489,end],
[_64474,_491756],
[_491756,nil],
[_64480,_491783],
[_491783,[[index,_63504],[restr,[[first,_63516],[rest,_63592]]]]],
[_64486,_491810],
              [_491810,[[eol,_63492],[list,_63492]]],
              [_64240,_486423],
              [_486423,t],
              [_64246,_486450],
              [_486450,end],
              [_64036,_486477],
              [_486477,t],
              [_63396,_486524],
              [_486524,[[index,_63504],
                        [restr,[[first,_63516],
                                [rest,[[first,[[arg,_63504],
                                               [relation,
                                                disj([accordingto,belligerent,
                                                      dead,disinterested,
                                                      eastbound,further,
                                                      here,just,little,
                                                      loc,northbound,of,
                                                      on,partitive,sideways,
                                                      southbound,speed,still,
                                                      supposedly,unable,
                                                      westbound,with])]]],
                                       [rest,_63592]]]]]]],
              [_63516,_488679],
              [_488679,[[instance,_63504],[relation,driver]]],
              [_63504,_488642],
              [_488642,[[gend,neuter],[num,singular],[per,third]]],
              [_63592,_488905],
              [_488905,end],
              [_64054,_486551],
              [_486551,[[eol,_63030],[list,_63492]]],
              [_63030,_487108],
              [_487108,t],
              [_63492,_488588],
              [_488588,t]]]])]],
[lex_rule([s(0),
           [glarf|_209272],
           [[[maj,adv],
             [pecParseTree,[[daughters,[[word,glarf]]],[rule_number,rn_1001]]],
             [phon,[[eol,_209273],[list,[[first,glarf],[rest,_494644]]]]],
             [synsem,[[local,[[category,[[bar,zero],
                                         [comps,end],
                                         [head,[[maj,adv],
                                                [mod,
                                                 [[synsem,
                                                  [[local,
                                                    [[content,
                                                      [[psoa,_209274],
                                                       [restr,_209275]]]]]]]]]],
                                        [subj,t]]],
                             [content,[[psoa,_494858],
                                       [restr,[[first,[[relation,_209276],
                                                       [soaarg,_495038]]],
                                               [rest,_494887]]]]]]]]],
           [[_209276,_495009],
            [_209275,_494887],
            [_209274,_209277],
            [_209277,_494858],
            [_209273,_494644],
            [_495009,t],
            [_494887,t],
            [_494858,_495038],
            [_495038,t],
            [_494644,t]]]]),
[lex_rule([s(0),
           [glarf|_209272],
           [[[maj,adv],
             [pecParseTree,[[daughters,[[word,glarf]]],[rule_number,rn_1001]]],
             [phon,_254205],
             [qstore,_254211],
             [synsem,[[local,[[category,[[bar,zero],
                                         [comps,_254451],
                                         [head,_254247],
                                         [marking,_254463],
                                         [spr,_254469],
                                         [subj,_254259]]],
                              [conj,nil],
                              [content,_253515],
                              [context,_254277]]]]]],
            [[_254205,_495373],
             [_495373,[[eol,_253869],[list,_253599]]],
             [_253599,_496519],
             [_496519,[[first,glarf],[rest,_253869]]],
             [_253869,_497523],
             [_497523,[[rest,_253593]]],
             [_253593,_496490],
             [_496490,t],
             [_254211,_495402],
             [_495402,[[eol,_253575],[list,_253581]]],
[_253575,_496411],
[_496411,t],
[_253581,_496440],
[_496440,t],
[_254451,_495477],
[_495477,end],
[_254247,_495506],
[_495506,[[maj,_254175],[mod,[[synsem,_253833]]]]],
[_254175,_498603],
[_498603,adv],
[_253833,_497383],
[_497383,[[local,[[category,[[bar,_253695],
                             [comps,_253701],
                             [head,_253375],
                             [marking,_253713],
                             [spr,_253719],
                             [subj,_253503]]],
                  [conj,_253731],
                  [content,_253983],
                  [context,_253989]]]]],
[_253695,_496830],
[_496830,one],
[_253701,_496859],
[_496859,end],
[_253375,_495706],
[_495706,[[form,finite],[maj,_253389]]],
[_253389,_495751],
[_495751,v],
[_253713,_496911],
[_496911,t],
[_253719,_496940],
[_496940,end],
[_253503,_496122],
[_496122,[[first,[[synsem,_253557]]],[rest,end]]],
[_253557,_496341],
[_496341,[[local,[[category,[[bar,two],
                             [comps,end],
                             [head,_254815],
                             [marking,_254821],
                             [spr,end],
                             [subj,_254833]]],
                  [conj,nil],
                  [content,_254127],
                  [context,[[eol,_253539],[list,_253539]]]]]]],
[_254815,_500664],
[_500664,[[case,nom],[form,norm],[maj,_254935]]],
[_254935,_501083],
[_501083,n],
[_254821,_500693],
[_500693,unmarked],
[_254833,_500738],
[_500738,end],
[_254127,_498411],
[_498411,[[index,_254875],
          [restr,[[first,[[instance,_254875],[relation,driver]]],
                  [rest,end]]]]],
[_254875,_500895],
[_500895,[[gend,neuter],[num,singular],[per,third]]],
[_253731,_496996],
[_496996,nil],
[_253983,_497883],
[_497883,[[restr,_254025],[tense,past]]],
[_253989,_497912],
[_497912,[[eol,_253533],[list,_253533]]],
[_254463,_495535],
[_495535,t],
[_254469,_495564],
[_495564,end],
[_254259,_495593],
[_495593,t],
[_253515,_495642],
[_495642,[[restr,[[first,[[relation,disj([accordingto,across,
                                          after,against,along,
                                          also,as,asfaras,at,
                                          before,behind,dest,
                                          dir,down,for,from,
                                          further,here,in,
                                          just,little,loc,
                                          not,of,on,over,past,
                                          path_change,sideways,
                                          simultaneouswith,
                                          still,suddenly,
                                          supposedly,then,
                                          through,toward,
                                          wednesday,when,
                                          while,with,within,
                                          without])]]],
                  [rest,_254025]]]]],
               [_254025,_498066],
               [_498066,end],
               [_254277,_495671],
               [_495671,[[eol,_253533],[list,_253539]]],
               [_253533,_496252],
               [_496252,t],
               [_253539,_496281],
               [_496281,t]]]])]]]
```

E.2.4 The driver stopped glarf

```
[[lex_rule([s(0),
            [glarf|_47614],
            [[[maj,adv],
              [pecParseTree,[[daughters,[[word,glarf]]],[rule_number,rn_1001]]],
              [phon,[[eol,_47615],[list,[[first,glarf],[rest,_343047]]]]],
              [synsem,[[local,[[category,[[bar,zero],
                                          [comps,end],
                                          [head,[[maj,adv],
                                                 [mod,
                                                  [[synsem,
                                                    [[local,
                                                      [[content,
                                                        [[psoa,_47616],
                                                         [restr,_47617]]]]]]]]]]],
                                          [subj,t]]],
                               [content,[[psoa,_343320],
                                         [restr,[[first,[[relation,_47618],
                                                         [soaarg,_343398]]],
                                                 [rest,_343431]]]]]]]]],
             [[_47618,_343369],
              [_47617,_343241],
              [_47616,_47619],
              [_47619,_343212],
              [_47615,_342992],
              [_343369,_343487],
              [_343487,t],
              [_343241,_343431],
              [_343431,t],
              [_343212,_343553],
              [_343553,_343320],
              [_343320,_343398],
            [_343398,t],
            [_342992,_343047],
            [_343047,t]]]]),
[lex_rule([s(0),
           [glarf|_47614],
           [[[maj,adv],
             [pecParseTree,[[daughters,[[word,glarf]]],
                            [rule_number,rn_1001]]],
             [phon,_81414],
             [qstore,_81420],
             [synsem,[[local,[[category,[[bar,zero],
                                         [comps,_81660],
                                         [head,_81456],
                                         [marking,_81672],
                                         [spr,_81678],
                                         [subj,_81468]]],
                              [conj,nil],
                              [content,_81108],
                              [context,_81486]]]]]],
            [[_81414,_343901],
             [_343901,[[eol,_81186],[list,_81564]]],
             [_81564,_344322],
             [_344322,[[first,glarf],[rest,_81186]]],
             [_81186,_344293],
             [_344293,t],
             [_81420,_343930],
             [_343930,[[eol,_81168],[list,_81174]]],
             [_81168,_344649],
             [_344649,t],
             [_81174,_344678],
             [_344678,t],
             [_81660,_344005],
             [_344005,end],
             [_81456,_344034],
             [_344034,[[maj,_81384],[mod,[[synsem,_81528]]]]],
             [_81384,_344941],
             [_344941,adv],
             [_81528,_344980],
             [_344980,[[local,[[category,[[bar,_81288],
                                          [comps,_81294],
                                          [head,_80968],
                                          [marking,_81306],
                                          [spr,_81312],
                                          [subj,_81096]]],
                  [conj,_81324],
                  [content,_81912],
                  [context,_81918]]]]],
[_81288,_345149],
[_345149,one],
[_81294,_345178],
[_345178,end],
[_80968,_345207],
[_345207,[[form,finite],[maj,_80982]]],
[_80982,_345609],
[_345609,v],
[_81306,_345236],
[_345236,t],
[_81312,_345265],
[_345265,end],
[_81096,_345294],
[_345294,[[first,[[synsem,_81150]]],[rest,end]]],
[_81150,_345882],
[_345882,[[local,[[category,[[bar,two],
                             [comps,end],
                             [head,_82408],
                             [marking,_82414],
                             [spr,end],
                            [subj,_82426]]],
                  [conj,nil],
                  [content,_81966],
                  [context,[[eol,_81132],[list,_81132]]]]]]],
[_82408,_346040],
[_346040,[[case,nom],[form,norm],[maj,_82528]]],
[_82528,_346351],
[_346351,n],
[_82414,_346069],
[_346069,unmarked],
[_82426,_346114],
[_346114,end],
[_81966,_346163],
[_346163,[[index,_82468],
          [restr,[[first,[[instance,_82468],[relation,driver]]],
                  [rest,end]]]]],
[_82468,_346614],
[_346614,[[gend,neuter],[num,singular],[per,third]]],
[_81324,_345327],
[_345327,nil],
[_81912,_345356],
```

373

```
            [_345356,[[restr,_81936],[tense,past]]],
            [_81918,_345385],
            [_345385,[[eol,_81132],[list,_81132]]],
            [_81132,_347049],
            [_81672,_344063],
            [_344063,t],
            [_81678,_344092],
            [_344092,end],
            [_81468,_344121],
            [_344121,t],
            [_81108,_344170],
            [_344170,[[restr,[[first,[[relation,disj([accordingto,across,
                                                      after,against,along,
                                                      also,as,asfaras,at,
                                                      before,behind,dest,
                                                      dir,down,for,from,
                                                      further,here,in,
                                                      just,little,loc,
                                                      not,of,on,over,past,
                                                      path_change,sideways,
                                                      simultaneouswith,
                                                      still,suddenly,
                                                      supposedly,then,
                                                      through,toward,
                                                      wednesday,when,
                                                      while,with,within,
                                                      without])]]],
                              [rest,_81936]]]]],
            [_81936,_346947],
            [_346947,end],
            [_81486,_344199],
            [_344199,[[eol,_81126],[list,_81132]]],
            [_81126,_347049],
            [_347049,t],
            [_81132,_346202],
            [_346202,t]]]]),
lex_rule([s(0),
          [glarf|_47614],
          [[[maj,adv],
            [pecParseTree,[[daughters,[[word,glarf]]],[rule_number,rn_1001]]],
            [phon,_100906],
            [qstore,_100912],
            [synsem,[[local,[[category,[[bar,zero],
                                        [comps,_101152],
                            [head,_100948],
                            [marking,_101164],
                            [spr,_101170],
                            [subj,_100960]]],
                 [conj,nil],
                 [content,_100468],
                 [context,_100978]]]]]],
[[_100906,_343901],
 [_343901,[[eol,_100546],[list,_101056]]],
 [_101056,_344322],
 [_344322,[[first,glarf],[rest,_100546]]],
 [_100546,_344293],
 [_344293,t],
 [_100912,_343930],
 [_343930,[[eol,_100528],[list,_100534]]],
 [_100528,_344649],
 [_344649,t],
 [_100534,_344678],
 [_344678,t],
 [_101152,_344005],
 [_344005,end],
 [_100948,_344034],
 [_344034,[[maj,_100876],[mod,[[synsem,_101020]]]]],
 [_100876,_344941],
 [_344941,adv],
 [_101020,_344980],
 [_344960,[[local,[[category,[[comps,_100642],
                             [head,_100328],
                             [marking,_100654],
                             [spr,_100660],
                             [subj,_100456]]],
                  [conj,_100828],
                  [content,_345356],
                  [context,_345385]]]]],
 [_100642,_345178],
 [_345178,end],
 [_100328,_345207],
 [_345207,[[form,finite],[maj,_100342]]],
 [_100342,_345609],
 [_345609,v],
 [_100654,_345236],
 [_345236,t],
 [_100660,_345265],
 [_345265,end],
[_100456,_345294],
[_345294,[[first,[[synsem,_100510]]],[rest,end]]],
[_100510,_345882],
[_345882,[[local,[[category,[[bar,two],
                             [comps,end],
                             [head,_101744],
                             [marking,_101750],
                             [spr,end],
                             [subj,_101762]]],
                  [conj,nil],
                  [content,_101464],
                  [context,[[eol,_100492],[list,_100492]]]]]]],
[_101744,_346040],
[_346040,[[case,nom],[form,norm],[maj,_101864]]],
[_101864,_346351],
[_346351,n],
[_101750,_346069],
[_346069,unmarked],
[_101762,_346114],
[_346114,end],
[_101464,_346163],
[_346163,[[index,_101804],
          [restr,[[first,[[instance,_101804],[relation,driver]]],
                  [rest,end]]]]],
[_101804,_346614],
[_346614,[[gend,neuter],[num,singular],[per,third]]],
[_100828,_345327],
[_345327,nil],
[_345356,[[restr,_101416],[tense,past]]],
[_345385,[[eol,_100492],[list,_100492]]],
[_100492,_347049],
[_101164,_344063],
[_344063,t],
[_101170,_344092],
[_344092,end],
[_100960,_344121],
[_344121,t],
[_100468,_344170],
[_344170,[[restr,[[first,[[relation,disj([accordingto,across,
                                          after,against,along,
                                          also,as,asfaras,at,
                                          before,behind,dest,
                                          dir,down,for,from,
                                          further,here,in,
                                                         just,little,loc,
                                                         not,of,on,over,past,
                                                         path_change,sideways,
                                                         simultaneouswith,
                                                         still,suddenly,
                                                         supposedly,then,
                                                         through,toward,
                                                         wednesday,when,
                                                         while,with,within,
                                                         without])]]],
                                 [rest,_101416]]]]],
               [_101416,_346947],
               [_346947,end],
               [_100978,_344199],
               [_344199,[[eol,_100486],[list,_100492]]],
               [_100486,_347049],
               [_347049,t],
               [_100492,_346202],
               [_346202,t]]]]])]]]
```

E.2.5 The vehicle stopped glarf

But the above entries all still believe that this word must be used with a subject of "the driver." An input with a different subject takes care of this.

```
[[lex_rule([s(0),
            [glarf|_47535],
            [[[maj,adv],
              [pecParseTree,[[daughters,[[word,glarf]]],[rule_number,rn_1001]]],
              [phon,[[eol,_47536],[list,[[first,glarf],[rest,_198558]]]]],
              [synsem,[[local,[[category,[[bar,zero],
                                          [comps,end],
                                          [head,
                                           [[maj,adv],
                                            [mod,
                                             [[synsem,
                                               [[local,
                                                 [[content,
                                                   [[psoa,_47537],
                                                    [restr,_47538]]]]]]]]]],
                                          [subj,t]]],
                               [content,[[psoa,_198827],
                                         [restr,[[first,[[relation,_47539],
                                                       [soaarg,_198903]]],
                                               [rest,_198936]]]]]]]]],
           [[_47539,_198876],
            [_47538,_198750],
            [_47537,_47540],
            [_47540,_198723],
            [_47536,_198505],
            [_198876,_198990],
            [_198990,_199160],
            [_199160,t],
            [_198750,_199024],
            [_199024,_198936],
            [_198936,t],
            [_198723,_199058],
            [_199058,_199090],
            [_199090,_199272],
            [_199272,_198827],
            [_198827,_198903],
            [_198903,t],
            [_198505,_199124],
            [_199124,_198558],
            [_198558,t]]]]),
[lex_rule([s(0),
           [glarf|_47535],
           [[[maj,adv],
             [pecParseTree,[[daughters,[[word,glarf]]],[rule_number,rn_1001]]],
             [phon,_81331],
             [qstore,_81337],
             [synsem,[[local,[[category,[[bar,zero],
                                         [comps,_81577],
                                         [head,_81373],
                                         [marking,_81589],
                                         [spr,_81595],
                                         [subj,_81385]]],
                              [conj,nil],
                              [content,_81025],
                              [context,_81403]]]]]],
            [[_81331,_199572],
             [_199572,[[eol,_81103],[list,_81481]]],
             [_81481,_199971],
             [_199971,[[first,glarf],[rest,_81103]]],
             [_81103,_199944],
             [_199944,t],
             [_81337,_199599],
[_199599,[[eol,_81085],[list,_81091]]],
[_81085,_200202],
[_200202,t],
[_81091,_200229],
[_200229,t],
[_81577,_199672],
[_199672,end],
[_81373,_199699],
[_199699,[[maj,_81301],[mod,[[synsem,_81445]]]]],
[_81301,_200482],
[_200482,adv],
[_81445,_200519],
[_200519,[[local,[[category,[[bar,_81205],
                             [comps,_81211],
                             [head,_80885],
                             [marking,_81223],
                             [spr,_81229],
                             [subj,_81013]]],
                  [conj,_81241],
                  [content,_81829],
                  [context,_81835]]]]],
[_81205,_200682],
[_200682,one],
[_81211,_200709],
[_200709,end],
[_80885,_200736],
[_200736,[[form,finite],[maj,_80899]]],
[_80899,_201118],
[_201118,v],
[_81223,_200763],
[_200763,t],
[_81229,_200790],
[_200790,end],
[_81013,_200817],
[_200817,[[first,[[synsem,_81067]]],[rest,end]]],
[_81067,_201381],
[_201381,[[local,[[category,[[bar,two],
                             [comps,end],
                             [head,_82325],
                             [marking,_82331],
                             [spr,end],
                             [subj,_82343]]],
                  [conj,nil],
                  [content,_81883],
                   [context,[[eol,_81049],[list,_81049]]]]]],
[_82325,_201535],
[_201535,[[case,nom],[form,norm],[maj,_82445]]],
[_82445,_201832],
[_201832,n],
[_82331,_201562],
[_201562,unmarked],
[_82343,_201605],
[_201605,end],
[_81883,_201652],
[_201652,[[index,_82385],
          [restr,[[first,[[instance,_82385]]],[rest,end]]]]],
[_82385,_202085],
[_202085,[[gend,neuter],[num,singular],[per,third]]],
[_81241,_200848],
[_200848,nil],
[_81829,_200875],
[_200875,[[restr,_81853],[tense,past]]],
[_81835,_200902],
[_200902,[[eol,_81049],[list,_81049]]],
[_81589,_199726],
[_199726,t],
[_81595,_199753],
[_199753,end],
[_81385,_199780],
[_199780,t],
[_81025,_199827],
[_199827,[[restr,[[first,[[relation,disj([accordingto,across,
                                          after,against,along,
                                          also,as,asfaras,at,
                                          before,behind,dest,
                                          dir,down,for,from,
                                          further,here,in,
                                          just,little,loc,
                                          not,of,on,over,past,
                                          path_change,sideways,
                                          simultaneouswith,
                                          still,suddenly,
                                          supposedly,then,
                                          through,toward,
                                          wednesday,when,
                                          while,with,within,
                                          without])]]],
                   [rest,_81853]]]]],
            [_81853,_202408],
            [_202408,end],
            [_81403,_199854],
            [_199854,[[eol,_81043],[list,_81049]]],
            [_81043,_203243],
            [_203243,t],
            [_81049,_201689],
            [_201689,t]]]]),
lex_rule([s(0),
          [glarf|_47535],
          [[[maj,adv],
            [pscParseTree,[[daughters,[[word,glarf]]],[rule_number,rn_1001]]],
            [phon,_100823],
            [qstore,_100829],
            [synsem,[[local,[[category,[[bar,zero],
                                        [comps,_101069],
                                        [head,_100865],
                                        [marking,_101081],
                                        [spr,_101087],
                                        [subj,_100877]]],
                             [conj,nil],
                             [content,_100385],
                             [context,_100895]]]]]],
           [[_100823,_203536],
            [_203536,[[eol,_100463],[list,_100973]]],
            [_100973,_203951],
            [_203951,[[first,glarf],[rest,_100463]]],
            [_100463,_203922],
            [_203922,t],
            [_100829,_203565],
            [_203565,[[eol,_100445],[list,_100451]]],
            [_100445,_204174],
            [_204174,t],
            [_100451,_204203],
            [_204203,t],
            [_101069,_203640],
            [_203640,end],
            [_100865,_203669],
            [_203669,[[maj,_100793],[mod,[[synsem,_100937]]]]],
            [_100793,_204442],
            [_204442,adv],
            [_100937,_204481],
            [_204481,[[local,[[category,[[comps,_100559],
                                         [head,_100245],
                             [marking,_100571],
                             [spr,_100577],
                             [subj,_100373]]],
                  [conj,_100745],
                  [content,_202362],
                  [context,_202460]]]]],
[_100559,_204638],
[_204638,end],
[_100245,_204667],
[_204667,[[form,finite],[maj,_100259]]],
[_100259,_204986],
[_204986,v],
[_100571,_204696],
[_204696,t],
[_100577,_204725],
[_204725,end],
[_100373,_204754],
[_204754,[[first,[[synsem,_100427]]],[rest,end]]],
[_100427,_205235],
[_205235,[[local,[[category,[[bar,two],
                             [comps,end],
                             [head,_101661],
                             [marking,_101667],
                             [spr,end],
                             [subj,_101679]]],
                  [conj,nil],
                  [content,_101381],
                  [context,[[eol,_100409],[list,_100409]]]]]]],
[_101661,_205387],
[_205387,[[case,nom],[form,norm],[maj,_101781]]],
[_101781,_205692],
[_205692,n],
[_101667,_205416],
[_205416,unmarked],
[_101679,_205461],
[_205461,end],
[_101381,_205510],
[_205510,[[index,_101721],
          [restr,[[first,[[instance,_101721]]],[rest,end]]]]],
[_101721,_205931],
[_205931,[[gend,neuter],[num,singular],[per,third]]],
[_100745,_204787],
[_204787,nil],
[_202362,[[restr,_101333],[tense,past]]],
[_202460,[[eol,_100409],[list,_100409]]],
[_101081,_203698],
[_203698,t],
[_101087,_203727],
[_203727,end],
[_100877,_203756],
[_203756,t],
[_100385,_203805],
[_203805,[[restr,[[first,[[relation,disj([accordingto,across,
                                          after,against,along,
                                          also,as,asfaras,at,
                                          before,behind,dest,
                                          dir,down,for,from,
                                          further,here,in,
                                          just,little,loc,
                                          not,of,on,over,past,
                                          path_change,sideways,
                                          simultaneouswith,
                                          still,suddenly,
                                          supposedly,then,
                                          through,toward,
                                          wednesday,when,while,
                                          with,within,without])]]],
                  [rest,_101333]]]]],
[_101333,_206216],
[_206216,end],
[_100895,_203834],
[_203834,[[eol,_100403],[list,_100409]]],
[_100403,_206997],
[_206997,t],
[_100409,_205549],
[_205549,t]]]]])]]]
```

Bibliography

[Atkins and Levin, 1991] Beryl Atkins and Beth Levin. Admitting impediments. In Uri Zernik, editor, *Lexical Acquisition: Exploiting On-Line Resources to Build a Lexicon.* Lawrence Erlbaum Associates, Publishers, Hillsdale, NJ., 1991.

[Bach, 1974] Emmon Bach. *Syntactic Theory.* Holt, Rinehart and Winston, Inc., 1974.

[Ballard and Stumberger, 1986] Bruce W. Ballard and Douglas E. Stumberger. Semantic acquisition in teli: A transportable, user-customized natural language processor. In *ACL Proceedings, 24th Annual Meeting,* 1986.

[Ballard, 1984] Bruce W. Ballard. The syntax and semantics of user-defined modifiers in a transportable natural language processor. In *COLING '84,* 1984.

[Barnett *et al.*, 1990] Jim Barnett, Kevin Knight, Inderjeet Mani, and Elaine Rich. Knowledge and natural language processing. *Communications of the ACM,* 33(8):50–71, August 1990.

[Bresnan, 1978] Joan W. Bresnan. A realistic transformational grammar. In Morris Halle, Joan W. Bresnan, and G. A. Miller, editors, *Linguistic Theory and Psychological Reality,* pages 1–59. MIT Press, Cambridge, Massachusetts, 1978.

[Bresnan, 1982] Joan Bresnan, editor. *The Mental Representation of Grammatical Relations.* MIT Press, Cambridge, Mass., 1982.

[Carpenter, 1989] Bob Carpenter. *Phrase meaning and categorial grammar.* PhD thesis, Centre for Cognitive Science, University of Edinburgh, Edinburgh, Scotland, 1989.

[Carter, 1989] David M. Carter. Lexical acquisition in the core language engine. In *Proceedings of the Fourth Conference of the European Chapter of the Association for Computational Linguistics*, pages 137–144, Manchester, UK, April 10th-12th 1989.

[Charniak and McDermott, 1985] Eugene Charniak and Drew McDermott. *Artificial Intelligence.* Addison-Wesley Publishing Company, 1985.

[Chomsky, 1965] Noam Chomsky. *Aspects of the Theory of Syntax.* MIT Press, Cambridge, Mass., 1965.

[Church *et al.*, 1991] Kenneth Church, William Gale, Patrick Hanks, and Donald Hindle. Using statistics in lexical analysis. In Uri Zernik, editor, *Lexical Acquisition: Exploiting On-Line Resources to Build a Lexicon.* Lawrence Erlbaum Associates, Publishers, Hillsdale, NJ., 1991.

[Clark and Clark, 1980] Eve V. Clark and Herbert H. Clark. When nouns surface as verbs. *Language*, 1980.

[Downing, 1977] Pamela Downing. On the creation and use of english compound nouns. *Language*, 53(4):810–842, 1977.

[Dowty, 1979] David R. Dowty. *Word Meaning and Montague Grammar.* D. Reidel Publishing Company., 1979.

[Dulay and Burt, 1974] Heidi C. Dulay and Marina K. Burt. You can't learn without goofing: An analysis of children's second language 'errors'. In Jack Richards, editor, *Error Analysis: Perspectives on Second Language Acquisition.* Longman, London, 1974.

[Dyer, 1991] Michael Dyer. Lexical acquisition through symbol recirculation in distributed connectionist networks. In Uri Zernik, editor, *Lexical Acquisition: Exploiting On-Line Resources to Build a Lexicon.* Lawrence Erlbaum Associates, Publishers, Hillsdale, NJ., 1991.

[Earley, 1970] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970. Also in Gross et al. (1986).

[Frisch *et al.*, 1990] Alan M. Frisch, Dale Gerdemann, John Griffith, Erhard W. Hinrichs, Jerry L. Morgan, Dale W. Russell, and Herry Sutanto. Automated natural language generation in a driving instruction system. In *Presented at the Motorola AI Applications Symposium*, Phoenix, AZ, November 9 1990.

[Gazdar and Mellish, 1989] Gerald Gazdar and Chris Mellish. *Natural Language Processing in PROLOG: An Introduction to Computational Linguistics.* Addison-Wesley Publishing Company, 1989.

[Gazdar *et al.*, 1985a] Gerald Gasdar, Ewan Klein, Geoffrey Pullum, and Ivan Sag. *Generalized Phrase Structure Grammar.* Harvard University Press, Cambridge, Mass., 1985.

[Gazdar *et al.*, 1985b] Gerald Gasdar, Geoffrey K. Pullum, R. Carpenter, Ewan Klein, T. Hukari, and R. Levine. Category structures. *Computational Linguistics*, 14:1–19, 1985.

[Gerdemann and Hinrichs, 1988] Dale Gerdemann and Erhard W. Hinrichs. Unicorn: A unification parser for attribute-value grammars. *Studies in the Linguistic Sciences*, 18(2):41–86, 1988.

[Gerdemann, 1991] Dale Gerdemann. *Parsing and Generation of Unification Grammars.* PhD thesis, University of Illinois at Urbana-Champaign, 1991. Also published as Technical Report CS-91-06 (Language Series) Cognitive Science Department, The Beckman Institute.

[Guo, 1987] Cheng-ming Guo. Interactive vocabulary acquisition in xtra. In *IJCAI*, volume 2, pages 715–717, 1987.

[Hastings *et al.*, 1991] Peter M. Hastings, Steven L. Lytinen, and Robert K. Lindsay. Learning words from context. In *Machine Learning 91*, 1991.

[Helmreich, 1985] Stephen C. Helmreich. Unpublished dissertation proposal. University of Illinois at Urbana-Champaign, 1985.

[Hirsh, 1990] Haym Hirsh. Incremental version-space merging. In *Proceedings of the Seventh International Conference on Machine Learning*, Austin, Texas, June 1990.

[Jackendoff, 1977] Ray Jackendoff. *X-bar Syntax: A Study of Phrase Structure.* MIT Press, 1977.

[Jacobs and Rau, 1988] Paul S. Jacobs and Lisa Rau. A friendly merger of conceptual expectations and linguistic analysis in a text processing system. In *Proceedings of the Fourth*

Conference on Artificial Intelligence Applications, pages 351–356, San Diego, CA, March 14-18 1988.

[Jacobs and Zernik, 1988] Paul S. Jacobs and Uri Zernik. Acquiring lexical knowledge from text: A case study. In *Proceedings of the Seventh National Conference on Artificial Intelligent Text-Based Systems*, pages 739–744, Minneapolis, MN, 1988.

[Kasman, 1991] Rick Kasman. Babel: A psychologically plausible cross-linguistic model of lexical and syntactic acquisition. In *Machine Learning 91*, 1991.

[Knight, 1989] Kevin Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, March 1989.

[Knuth, 1965] Donald E. Knuth. On the translation of languages from left to right. *Information and control 8*, pages 607–639, 1965.

[Kornai and Pullum, 1990] Andras Kornai and Geoffrey K. Pullum. The x-bar theory of phrase structure. *Language*, 66(1):24–50, 1990.

[Laird *et al.*, 1987] John E. Laird, A. Newell, and P.S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[Lake, 1991] J. Michael Lake. PEC user's manual: A path equation compiler for Unicorn-3. Technical Report CS-91-11, Beckman Institute for Advanced Science and Technology, 1991.

[Lang and Hirschman, 1988] Francois-Michel Lang and Lynette Hirschman. Improved portability and parsing through interactive acquisition of semantic information. In *2nd Applied Natural Language Processing, ACL*, 1988.

[Lenat and Guha, 1990] Douglas B. Lenat and R.V. Guha. *Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project.* Addison-Wesley Publishing Company, 1990.

[Lytinen and Hastings,] Steven L. Lytinen and Peter M. Hastings. Acquiring new words from context. Submitted to *Machine Learning.*

[Lytinen, 1992] Steven L. Lytinen. A unification-based, integrated natrual language processing system. *Computers and Mathematics with Applications*, 23(6-9):403–418, 1992.

[Martin, 1988] James H. Martin. *A Computational Theory of Metaphor*. PhD thesis, University of California, Berkeley, 1988. Computer Science Department, Berkeley, CA, 1988. Report No. UCB/CSD 88-465.

[Miller and Laird, 1991] Craig S. Miller and John E. Laird. A constraint-motivated model of lexical acquisition. In *Machine Learning 91*, 1991.

[Mitchell, 1977] Tom M. Mitchell. Version spaces: A candidate elimination approach to rule learning. In *Proceedings of the 1977 IJCAI*, pages 305–310, Cambridge, MA, 1977.

[Nicholl, 1991] Sheldon Nicholl. *Language Acquisition by Computer: Categories, Agreement, and Morphology*. PhD thesis, University of Illinois at Urbana-Champaign, 1991.

[Nunberg, 1978] Geoffrey D. Nunberg. *The Pragmatics of Reference*. PhD thesis, University of California, Berkeley, Indiana University Linguistics Club, Bloomington IN, 1978.

[Pollard and Sag, 1987] Carl Pollard and Ivan A. Sag. *Information Based Syntax and Semantics, Volume 1: Fundamentals.*, volume CSLI Lecture Note Series No. 13. Stanford University: Center for the Study of Language and Information. Distributed by University of Chicago Press., 1987.

[Pollard and Sag, In press] Carl Pollard and Ivan A. Sag. *Agreement, Binding and Control: Information-Based Syntax and Semantics, Volume 2*. Stanford University: Center for the Study of Language and Information. Distributed by University of Chicago Press., In press.

[Pollard, 1989] Carl Pollard. The syntax-semantics interface in a unification-based phrase structure grammar. The Valentine's Day Paper, Feb. 24 1989.

[Pullum, 1985] Geoffrey K. Pullum. Assuming some version of x-bar theory. In *Chicago Linguistic Society*, volume 21, pages 323–353, 1985.

[Rayner *et al.*, 1988] Manny Rayner, Asa Hugosson, and Goran Hagert. Using a logic grammar to learn a lexicon. In *COLING 88*, 1988.

[Russell, 1987] Dale Russell. *Cheyenne Verb Agreement in the Theory of Generalized Phrase Structure Grammar*. PhD thesis, University of Illinois at Urbana-Champaign, 1987.

[Schank and Abelson, 1977] Roger C. Schank and R. P. Abelson. *Scripts, Plans, Goals, and Understanding.* Lawrence Erlbaum, Hillsdale, N.J., 1977.

[Schank, 1975] Roger C. Schank. *Conceptual Information Processing.* North-Holland, New York, 1975.

[Shieber, 1985] Stuart M. Shieber. Using restriction to extend parsing algorithms for complex-feature-based formalisms. In *ACL Proceedings, 23rd Annual Meeting*, pages 145–152, 1985.

[Siskind, 1990] Jeffrey Mark Siskind. Acquiring core meanings of words, represented as jackendoff-style conceptual structures, from correlated streams of linguistic and non-linguistic input. In *ACL Proceedings, 28th Annual Meeting*, 1990.

[Slator, 1991] Brian M. Slator. Using context for sense preference. In Uri Zernik, editor, *Lexical Acquisition: Exploiting On-Line Resources to Build a Lexicon.* Lawrence Erlbaum Associates, Publishers, Hillsdale, NJ., 1991.

[Trost and Buchberger, 1986] H. Trost and E. Buchberger. Towards the automatic acquisition of lexical data. In *COLING 86*, 1986.

[Wasow *et al.*, 1983] Thomas Wasow, Ivan A. Sag, and Geoffrey Nunberg. Major and minor rules in lexical grammar. In Teun Hoekstra, Harry van der Hulst, and Michael Moortgaart, editors, *Lexical Grammar*, pages 285–312. Dordrecht: Foris, 1983.

[Webster and Marcus, 1989] Mort Webster and Mitch Marcus. Automatic acquisition of the lexical semantics of verbs from sentence frames. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, pages 177–184, 1989.

[Weizenbaum, 1966] J. Weizenbaum. Eliza – a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9:36–45, 1966.

[Zernik and Dyer, 1986] Uri Zernik and Michael G. Dyer. Disambiguation and language acquisition through the phrasal lexicon. In *COLING 86*, 1986.

[Zernik, 1988] Uri Zernik. Language acquisition: Coping with lexical gaps. In *COLING 88*, pages 796–800, 1988.

[Zernik, 1991] Uri Zernik. Introduction. In Uri Zernik, editor, *Lexical Acquisition: Exploiting On-Line Resources to Build a Lexicon.* Lawrence Erlbaum Associates, Publishers, Hillsdale, NJ., 1991.

Vita

Dale W. Russell was born on July 7, 1959 in Beaver Falls, Pennsylvania. He attended Geneva College in Beaver Falls, where in 1981, he received a B.A. degree in French with a minor in Speech Pathology, graduating summa cum laude.

In the fall of 1981 he entered the graduate program in Linguistics at the University of Illinois at Urbana-Champaign. There he was awarded a University Fellowship for two years, and a Teaching Assistantship for two years, teaching psycholinguistics. He was then awarded Research Assistantships for two years, working first on a bibliography of Amerindian languages with Prof. Rudolph Troike, and then on an Artificial Intelligence project in Natural Language Processing, with Prof. LaRaw Maran. He received the A.M. degree in Linguistics in 1983 and the Ph.D. in 1986, for work on the application of the principles of Generalised Phrase Structure Grammar to the morphological phenomena of Cheyenne, an American Indian language. Following graduation he spent a year doing post-doctoral work in computational linguistics at the University of Illinois.

In the fall of 1988 he began his graduate studies in Computer Science at the University of Illinois at Urbana-Champaign. He was awarded a Teaching Assistantship, and taught two introductory courses in Computer Science. In 1989, he joined the Knowledge-Based Systems Group and began investigating issues of Natural Language Processing from a knowledge-based perspective. In 1990, he was awarded a Research Assistantship, working with Profs. Jerry Morgan, Georgia Green, Erhard Hinrichs and Alan Frisch, on a project sponsored by Motorola, Inc., for an automated driving instruction system. For two years, he was awarded a Beckman Institute Research Assistantship. In August 1993, he defended a doctoral dissertation which focused on lexical acquisition in unification-based grammar, using a real-world knowledge base.

BIBLIOGRAPHIC DATA SHEET

1. Report No. UIUCDCS-R-93-1822

2.

3. Recipient's Accession No.

4. Title and Subtitle

Language Acquisition in a Unification-Based Grammar
Processing System Using a Real-World Knowledge Base

5. Report Date July 1993

6.

7. Author(s)

Dale W. Russell

8. Performing Organization Rept. No. R-93-1822

9. Performing Organization Name and Address

Department of Computer Science
University of Illinois
1304 W. Springfield Ave.
Urbana, IL 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.

12. Sponsoring Organization Name and Address

13. Type of Report & Period Covered

Ph.D. thesis

14.

15. Supplementary Notes

16. Abstracts

One of the obstacles to be overcome in Natural Language Understanding is the existence of lexical gaps; that is, words or word senses which are not in the lexicon of the system. No lexicon, whether hand-coded or derived from an on-line dictionary, can ever be complete, in the sense of having entries for every word encountered in every syntactic category and with every semantic sense with which it may be used. In order to address this issue, this thesis describes the implementation of MURRAY, a learning mechanism which is able to (i) infer the syntactic properties of a new lexical item from its syntactic environment; (ii) infer the meaning of a novel lexical item based on context and a domain-specific database of real-world knowledge; and (iii) combine those syntactic and semantic properties of a given unknown word inferred from multiple pieces of input, resulting in a version space of possible lexical entries for the unknown, each consistent with all environments in which it has been encountered. MURRAY is an extension to an existing unification-based grammar processing system, UNICORN. It has been implemented to operate with grammars written in the style of Head-Driven Phrase Structure Grammar, though it is compatible with any unification-based grammar formalism. On each encounter with a word which does not exist in the lexicon of the system, MURRAY constructs a lexical version hyperspace, a disjunction of version spaces, one for each possible syntactic category that the unknown could be in the given linguistic context. On each encounter with the unknown, information from the version spaces thus constructed is combined, so that from multiple inputs, the system converges on the target definition of the new word.

17. Key Words and Document Analysis. 17a. Descriptors

natural language processing
lexical acquisition
unification-based grammar

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement

unlimited

19. Security Class (This Report) UNCLASSIFIED

20. Security Class (This Page) UNCLASSIFIED

21. No. of Pages 390

22. Price

FORM NTIS-35 (10-70)

USCOMM-DC 40329-P71